Computer Science Research Review 1970-71

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Carnegie-Mellon University Dept of Computer Science Pittsburgh, Penn 15213	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

COMPUTER SCIENCE RESEARCH REVIEW 1970-71

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Scientific Interim

5. AUTHOR(S) *(First name, middle initial, last name)*

T[illegible] Moran

6. REPORT DATE 1970-71	7a. TOTAL NO. OF PAGES 67	7b. NO. OF REFS 0

8a. CONTRACT OR GRANT NO. F44620-70-C-0107	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO. A0827-5 c. 61102F d. 681304	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* AFOSR - TR - 72 - 0462

10. DISTRIBUTION STATEMENT

Approved for public release;
distribution unlimited.

11. SUPPLEMENTARY NOTES TECH, OTHER	12. SPONSORING MILITARY ACTIVITY Air Force Office of Scientific Research (NM) 1400 Wilson Blvd Arlington, Virginia 22209

13. ABSTRACT

This is the annual report published by the Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn. The reporting period is from 1970-1971. The series of papers includes A brief primer on Resolution Proof Procedures by Donald W. Loveland, Control Structures by David A. Fisher, Bliss: A Language for Programming Systems by William A. Wulf and the Kernal Approach to Building Software Systems by Allen Newell, Peter Freeman, Donald McCracken, and George Robertson.

Computer Science Research Review 1970-71

An Annual Report
published by the
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Edited by Tom Moran

The work reported here was largely supported by the
Advanced Research Projects Agency
of the Office of the Secretary of Defense
(Contract number F-44620-70-C-0107)
and is monitored by the
Air Force Office of Scientific Research.
This work was also supported in part by the following:
the National Institute of Health (MH-077722-02),
the National Science Foundation (GP-7064), and
the Mellon family and the Richard King Mellon Foundation.

Pittsburgh, Pennsylvania
September, 1971

Contents

Annual Review Introduction

This year's annual Review provides another sample of research in computer science at Carnegie-Mellon. Like our previous samples it is both revealing and concealing. In terms of content this issue's emphasis on system building systems (the paper by Bill Wulf and also by Newell *et al.)* is certainly revealing of a real emphasis in the CMU environment—an emphasis that David Fisher's paper on control structures broadens a bit, but fundamentally reinforces. Donald Loveland's paper suggests that our research is broader than this, not only by representing a more mathematical concern, but also by indicating a connection with artificial intelligence. But this entire sample, taken all together, still conceals the major growth in activity that has occurred this last year in the design of computer hardware systems and in the development of speech recognition systems.

There are more basic ways in which this annual Review is both revealing and concealing. When placed as the latest member of a sequence of such Reviews, it reveals a continued and invariant commitment to the development of a computer science. It reveals, thus, an unchanging aspect of this environment. But it thereby conceals that we have just undergone our most major organizational change since we became a Department of Computer Science in 1965. Alan Perlis has left to become the Higgins Professor of Computer Science at Yale University. Much of the outside world has identified computer science at CMU with Alan Perlis. Still, only those who have been in this environment can appreciate how thoroughly it was saturated with his presence. We wish him well at Yale. He claims to be finished with department-heading and such like occupations. We also wish him well in this resolve.

Joseph Traub has joined us as the new head of the Department of Computer Science, having previously been at the University of Washington in Seattle and before that at Bell Laboratories. His principle research interest is in numerical mathematics, an area in which we have not previously laid much emphasis. We welcome him and look forward to a continued strengthening and broadening of the research that has been illustrated by these annual Reviews.

A.N.
18 Aug 71

ZE

XA

A Brief Primer on Resolution Proof Procedures

Donald W. Loveland

For mechanical theorem proving the decade of the sixties has been dominated by the development of complete proof procedures of first order logic. Moreover, this approach has itself been dominated by the off-shoots of one complete procedure, *resolution*. A *complete proof procedure* in first order logic is a mechanical procedure which can always verify, given sufficient time and resources, that a given first order theorem is a theorem. Complete procedures cannot identify all non-theorems as such. This article contains a summary of the resolution-based complete procedures (here called resolution strategies). Special strategies for handling the equality relation, though important, have been excluded here.

The primary purpose of an article written for the *Computer Science Research Review* of CMU is to portray active areas of faculty research. Resolution-type proof procedures are of interest to the author[3-6] and to Professor Peter Andrews[1-2] of the CMU Mathematics Department. The author believes that the best way to present the results of this interest is to present a general picture into which the results fit.

A second purpose of this article is to fill a present gap in the survey literature. Until now there has been no article which a computer scientist (including a student) with a one semester logic background can read for a brief technical survey of basic resolution theory, although the word "resolution" is in almost every computer science student's vocabulary. This article is structured to provide enough information (without proof) to allow a person to design a reasonable resolution theorem-prover with an awareness of the several alternate approaches available to him.

The article is split into sections with the intent that some will be read more closely than others depending on the reader's background and interest. In particular, section 1 concerning the preparation of a formula for input to a resolution procedure can be omitted by anyone with some exposure to the literature of this area. The work at CMU (related to first order resolution theory) is contained within section 5. The only references to the people responsible for the ideas reported here are in section 7.

Almost all of the papers so far published containing results cited in this paper are referenced in Meltzer's paper.[10] To a lesser extent the references appear also in the Anderson and Bledsoe paper.[8] Also listed are J. A. Robinson's basic paper on resolution,[12] two papers of interest not referenced in Meltzer's paper,[7,9] a basic logic text[11] where the logical concepts needed for this paper can be found, and the relevant papers written at CMU.[1-6] For further study, the Robinson paper followed by the Anderson and Bledsoe paper is appropriate for those interested in the completeness aspects of these proof procedures, while the Meltzer paper is more appropriate for those less inclined in this direction.

1. Preparing a Formal Expression

The sentence to be tested for theoremhood is written in the (first order) predicate calculus. Preparation consists of two parts: formation of a first order sentence, or closed formula, which expresses the intuitively conceived theorem and then conversion of the sentence to proper input form. Often the first part is partially completed by drawing on standard knowledge of the field in question. This part can be quite difficult to execute successfully for many reasons. The second part is a straightforward algorithmic procedure. A brief discussion will be given of the nature of the formation of a first order sentence, chiefly by example, followed by a procedure for preparation of a sentence for input to a resolution proof procedure.

A proposed theorem is often presented as a set of hypotheses about an environment (call such statements *axioms)* implying some desired *assertion.* The chief difficulty in formulating such a sentence (we assume hereafter all sentences, or formulas, are assumed to be in first order logic) is to describe sufficiently the predicates introduced by axioms so as to disallow unintended interpretations. This cannot be done in general, as any student of logic knows, but in sufficiently simple cases it can be accomplished. It is usually the case, anyway, that a finite set of axioms exist which sufficiently describes the environment with respect to the "crucial properties" to allow a proof of any given "true" formula. However, these crucial properties are not usually known in advance of the proof, so one usually models one's intuition and tests the results.

Consider first a quite simple-minded real world example, the question if, in a particular environment, a monkey can get some bananas suspended from a ceiling of a room. Although the resulting theorem is trivial for most theorem provers, the process of formation of the statement may be instructive. Figure 1 lists the chosen constants and predicates (with their intended interpretation) plus the axioms in the final input form. The input form is developed later in this section. How might a set of formulas which generate the axioms of Figure 1 be chosen?

Figure 1.
Monkey-banana problem.

Constants:

M	= monkey
B	= bananas
C	= chair
F	= floor

Predicates:

reach(x,y)	= x can reach y
get(x,y)	= x can get y
andex(x)	= x is a dexterous animal
close(x,y)	= x is close to y
on(x,y)	= x is on y (x can get on y)
under(x,y)	= x is under y
tall(x)	= x is tall (x is high)
inroom(x)	= x is in the (given) room
move(x,y,z)	= x can move y near z
climb(x,y)	= x can climb y

Axioms:

(i) $\sim$ reach(x,y) get(x,y)

(ii) $\sim$ andex(x) $\sim$ close(x,y) reach(x,y)

(iii) $\sim$ on(x,y) $\sim$ under(y,B) $\sim$ tall(y) close(x,B)

(iv) $\sim$ inroom(x) $\sim$ inroom(y) $\sim$ inroom(z) $\sim$ move(x,y,z) close(z,F) under(y,z)

(v) $\sim$ climb(x,y) on(x,y)

(vi) andex(M)

(vii) tall(C)

(viii) inroom(M)

(ix) inroom(B)

(x) inroom(C)

(xi) move(M,C,B)

(xii) $\sim$ close(B,F)

(xiii) climb(M,C)

Assertion: get (M,B)

The assertion to be tested is whether or not the monkey gets the bananas. Suppose the setting is a room with the bananas suspended in a corner away from both the monkey and a light chair. The monkey on the chair would be tall enough to get to the bananas. The goal is "get(M,B)" which is named as the assertion. Suppose the word "reach" means physical proximity plus outstretched arm in the proper direction. Then a natural choice of axiom is:

(1) $\mathrm{reach}(x,y) \supset \mathrm{get}(x,y)$

That is, for all x,y, if x reaches for y, x gets y. Note the implicit universal quantification of x and y; this is traditional and adopted here. The predicate "reach" has additional strong connotations which must be sufficiently described by one or more further axioms:

(2) $[\mathrm{andex}(x)\ \&\ \mathrm{close}(x,y)] \supset \mathrm{reach}(x,y)$

This constrains the predicate "reach" but only by introducing two unconstrained (or undefined) predicates. It is reasonable simply to assert "andex(M)" (see axiom vi). More axioms could be provided concerning the predicate "andex" to better specify the situation in hopes that this would help establish the assertion. We refrain from doing so here to keep the example of modest size. By hindsight it is known that such extra axioms actually are not needed in this case. The predicate "close" is constrained by the description (definition):

(3) $[\mathrm{on}(x,y)\ \&\ \mathrm{under}(y,B)\ \&\ \mathrm{tall}(y)] \supset \mathrm{close}(x,B)$.

Continuing in this manner, "under" and "on" are themselves constrained:

(4) $[\mathrm{inroom}(x)\ \&\ \mathrm{inroom}(y)\ \&\ \mathrm{inroom}(z)\ \&\ \sim \mathrm{close}(z,F)] \supset [\mathrm{move}(x,y,z) \supset \mathrm{under}(y,z)]$;

(5) $\mathrm{climb}(x,y) \supset \mathrm{on}(x,y)$.

These constraints establish the need for additional predicates, but at this stage it is not unreasonable to define them by specific identification of object with property as was done with "andex". Hence, add axioms (vi) through (xiii) of Figure 1. As stated before, the assertion about the environment is

(6) get(M,B).

The candidate theorem is the *universal closure* of the *conjunction* of formulas (1) through (6) and axioms (vi) through (xiii). By conjunction is meant connecting the formulas by &'s and by universal closure is meant adding sufficient universal quantifiers around the outside of the total formula to leave no free variables.

As an example of a mathematical system that is described by a finite number of axioms, a natural candidate is group theory. Let P(x,y,z) be interpreted as $x \cdot y = z$. Then a group is a structure which satisfies the following axioms (recall axioms are equivalent to their universal closure):

(7) $\exists z P(x,y,z)$ *closure*

(8) $[P(x,y,u)\ \&\ P(y,z,v)] \supset [P(x,v,w) \equiv P(u,z,w)]$ *associativity*

(9) $\exists x[\forall y\, P(x,y,y)\ \&\ \forall y \exists z P(z,y,x)]$ *Left identity and Left inverse*

The statement $(7^c)\ \&\ (8^c)\ \&\ (9^c) \supset \exists x[\forall y P(x,y,y)\ \&\ \forall y \exists z P(y,z,x)]$, where, e.g., (7^c) denotes the universal closeure of axiom (7), illustrates a reasonable formula to test for theoremhood. The intended interpretation is that right inverses exist for each element of a group. The input form of a theorem slightly stronger than this one will be given later.

Elementary number theory may be the most important of all axiomatic mathematical theories, but its natural axiomization involves an infinite number of first order axioms. A large portion of elementary number theory can be handled within a finite, and indeed relatively small, number of axioms, but in such formulations even the simplest results of number theory seem beyond the present procedures when reasonable time limits are imposed. This is one of the reasons why a different approach to theorem-proving may dominate the next decade. However, this will not automatically be the case. Heuristics superimposed on resolution strategies may outperform even supposedly preferable systems, just as the refinements to the internal combustion engine has to date allowed it to dominate the steam engine in practice.

Of course, formulas simply to be tested for logical validity (or unsatisfiability) are suitable inputs also. For example, one might wish to establish that the following formula is valid:

(10) $\forall x[A(x) \supset B(x)] \supset [\exists xA(x) \supset \exists xB(x)]$.

Here quantifiers take the smallest scope consistent with the parenthesizing. This is also true for negation symbols when used.

The proof procedures under discussion are capable of indicating that certain formulas are unsatisfiable. Therefore, the formulas actually used for input to the resolution procedures are chosen so that unsatisfiability is the quality to be ascertained. Thus, a formula to be tested for validity is negated to yield a formula to be tested for unsatisfiability, the second formula being unsatisfiable if and only if the original formula is valid.

Steps 0 through 7 below describe how to convert a closed formula to an appropriate input form. The input formula is said to be in *Skolem functional form* when it is in the format obtained at the end of step 7.

Assume the formula presented for testing is closed. If the formula has free variables, and the test is for validity, add to the left of the formula a universal quantifier of each free variable. The order of attachment is unimportant. For example, the formula $A(x) \supset A(y)$ should be replaced by $\forall x \forall y(A(x) \supset A(y))$. If the test is for unsatisfiability, add existential quantifiers in like manner.

Step 0. Obtain unsatisfiable formula. If the given formula is to be tested for validity, negate the formula. If the test is for unsatisfiability, do not negate.

Example: Test equation (10) for validity. Thus negate (10) giving:

(11) $\sim [\forall x[A(x) \supset B(x)] \supset [\exists xA(x) \supset \exists xB(x)]]$

Step 1. Eliminate $\equiv$ *and* $\supset$. Change each occurrence of $R \equiv S$ to $(\sim R \vee S) \,\&\, (\sim S \vee R)$. Change each occurrence of $R \supset S$ to $\sim R \vee S$.

Applying this step to (11) gives:

(12) $\sim[\sim\forall x[\sim A(x) \vee B(x)] \vee [\sim \exists xA(x) \vee \exists xB(x)]]$.

Step 2. Rename variables. Change names of bound variable occurrences when necessary so that each variable appears in one quantifier. Any change of name, of course, must be uniform throughout the scope of a quantifier. The renaming of variables is necessary to make the transformations of step 3 acceptable.

Example (cont'd): From (12):

(13) $\sim [\sim \forall x[\sim A(x) \vee B(x)] \vee [\sim\exists yA(y) \vee \exists zB(z)]]$.

Step 3. Place in prenex normal form. Replace the formula obtained after step 2 by another having the quantifiers all moved to the left as far as possible using only the following transformations (here A and B are arbitrary formulas and x denotes an arbitrary variable):

(a) $\sim \forall xA$ to $\exists x \sim A$
$\sim \exists xA$ to $\forall x \sim A$;

(b) $\forall xA \vee B$ to $\forall x(A \vee B)$, $A \vee \forall xB$ to $\forall x(A \vee B)$,
$\exists xA \vee B$ to $\exists x(A \vee B)$, $A \vee \exists xB$ to $\exists x(A \vee B)$;

(c) As (b), with & everywhere replacing V.

Notice that $\forall x(\exists yA \vee B)$ can go to $\forall x\exists y(A \vee B)$ but not to $\exists y\forall x(A \vee B)$. Thus quantifier order is important. However, $\exists yA \vee \forall zB$ can go to either $\exists y\forall z(A \vee B)$ or $\forall z\exists y(A \vee B)$. Within the range of permissible moves, it is very desirable to bring each existential quantifier to the left of as many universal quantifiers as possible. The reason will become evident later.

Example (cont'd.): From (13):

(14) $\exists y\forall z\forall x \sim[\sim[\sim A(x) \vee B(x)] \vee [\sim A(y) \vee B(z)]]$

Step 4. Eliminate ∃. Replace all existentially quantified variable occurrences by *Skolem functions.* For each existentially quantified varaible, introduce an expression, called a (Skolem) *function instance,* consisting of a new function letter and, as arguments, all variables which have universal quantifiers to the left of the pertient existential quantifier. A new constant, i.e., 0-ary function, is associated with any existentially quantified variable whose quantifier precedes all universal quantifiers. After all appropriate function instances are formed, drop all quantifiers and then for every variable occurrence which has an associated function instance (i.e., any variable previously existentially quantified) replace the variable by its associated function instance. For example, ∃x∀y∃z∀wP(x,y,z,w) becomes P(a,y,f(y),w) where P is a quantifier free formula not containing the constant a or the function symbol f.

Example (cont'd.): From (14):

(15) ~[~[~A(x) V B(x)] V [~A(a) V B(z)]],
where a is a constant.

Step 5. Place in conjunctive normal form. To define *conjunctive normal form* it is convenient to introduce some useful terminology. An *atomic formula,* or *atom,* is a predicate instance, i.e., a predicate letter followed by its arguments, which are terms. A *literal* is an atom or its negation. A *disjunctive clause*, or simply *clause*, is a disjunction of literals. For example, ~ P(x) V Q(x,y) V ~ R(y) is a clause of three literals. A quantifier-free formula (qff) is in *conjunctive normal form* (cnf) if it is the conjunction of clauses. Thus P(x) & ~ Q(y) & (R(y) V ~ R(x) is in cnf. A qff is placed in cnf by appropriate use of the following transformations:

(a) ~ (AVB) to ~ A & ~ B, ~ (A & B) to ~ A V ~ B,
(b) A V (B & C) to (AVB) & (AVC),
(c) ~ ~ A to A.

Recall that V and & are commutative and associative.

Example (cont'd.): A sequence of transformations takes (15) into a formula (16) in cnf.

```
     ~ ~ [~A(x) V B(x)] & ~ [~A(a) V B(z)]   by (a)
         [~A(x) V B(x)] & ~ [~A(a) V B(z)]   by (c)
         [~A(x) V B(x)] & ~ ~A(a) & ~B(z)    by (a)
(16)     [~A(x) V B(x)] & A(a) & ~B(z)       by (c)
```

Step 6. Simplify. If a formula from step 5 contains a clause with both an atom and its negation, the clause may be eliminated. A second occurrence of a literal in a clause may also be eliminated. If a clause C exists which contains all the literals of another clause and perhaps more, then clause C can be eliminated. (Stronger than this last condition is the rule that any subsumed clause can be eliminated, but it is desirable to defer the definition of clause subsumption.) As an example of simplification,

P(x) & [Q(x) V ~ P(y) V Q(x)] & [R(x) V ~ R(x)] & [P(x) V Q(y)]

can be simplified to

P(x) & [Q(x) V ~ P(y)].

Example (cont'd.): step 6 introduces no change on (16).

Step 7. Reduce notation. We shift to a set notation from a formal language notation and take advantage of the commutivity and associativity of & and V. Each clause is treated as a set of literals; the V symbol is dropped. However, the standard brace couplet { } denoting the boundary of the membership list is omitted. Literals of the same clause are simply listed adjacent to one another with no separating symbol. It is convenient to omit parentheses and commas from atoms also, writing Rf(xy)xy for R(f(x,y),x,y), for example. Thus, for instance, the clause R(g(x),y) V Q(a,f(a,y)) is rewritten Rg(x)y Qaf(ay). The formula itself is altered to read as a set of clauses. The bracket delimiters are sometimes used with clauses separated by commas. Often formulas are presented by displaying one clause per line with the clauses on successive lines and the bracket delimiters omitted.

Example (cont'd.): (16) becomes:

(17) ~ Ax Bx
(18) Aa
(19) ~ Bz

A formula presented as given at step 6 or step 7 is said to be in *Skolem functional form.* The manipulations of steps 1, 2, 3, and 5 do not alter unsatisfiability because logical equivalences are involved. The introduction of Skolem functions at step 4 preserves only the property of having no model (unsatisfiability) or at least one model. The Skolem function can be regarded as the verifying function for the existential quantifier making the formula true (in an interpretation) if and only if there exists an individual making it true. The individual would clearly depend on those variables universally quantified within whose scope the existential quantifier lies.

A theorem within a first order theory with a finite number of axioms is easily converted to Skolem functional form as each axiom may be converted individually. Then the resultant clauses plus the negation of the theorem statement forms the appropriate formula for input. Thus the list of thirteen axioms and the negation of the assertion in Figure 1 presents the monkey-banana problem in input form. The group axioms (7), (8) and (9) in Skolem functional form are:

(20)	Pxyf(xy)	from (7)
(21)	~ Pxyu ~ Pyzv ~ Pxvw Puzw	from (8)
(22)	~ Pxyu ~ Pyzv ~ Puzw Pxvw	from (8)
(23)	Peyy	from (9)
(24)	Pg(y)ye	from (9)

It suffices to add the Skolem functional form of the negation of the assertion of a group theory theorem (and perhaps special hypotheses as axioms) to present a group theory problem. As an example consider

(25) ~ Pag(a)e

which arises from the negation of ∀xPxg(x)e. ∀xPxg(x)e asserts that the left inverse function g (introduced as a Skolem function) is also a right inverse function.

Sometimes a more desirable conversion can be obtained by introducing Skolem functions before obtaining a prenex normal form.

2. The Basic Operation of Resolution

Resolution can be characterized as the *cut* inference rule (or a generalized *modus ponens* inference rule) of propositional logic with an appropriate substitution rule. Alternately, one could say resolution is based on the tautology ((AVC) & (~ CVB) ⊃ AVB). Consider the propositional part of resolution first. Two literals are *complementary* if one differs from the other only by possession of a negation sign. Px and ~ Px are complementary literals. At the propositional level the resolution operation takes two *parent* clauses $A_1A_2 \ldots A_n$ and $B_1B_2 \ldots B_m$ sharing a complementary pair of literals, the *literals resolved upon*, say $B_1 = \sim A_1$, and yields the *resolvent* clause $A_2A_3 \ldots A_nB_2 \ldots B_m$. (For formula notation see section 1, step 7). That is, the literal resolved upon in each clause is deleted and the remaining literals comprise the resolvent. It is always assumed that no two literals of a clause are identical. Thus B_k, $k \geq 2$, is omitted from explicit appearance if $B_k = A_i$, for some $i \geq 2$.

Substitution is used to form a complementary pair between two clauses when none exists explicitly. For example, PxRu and ~ Pf(y)Qu are two clauses not qualified for resolution as given above, but by replacing the x by f(y) clauses Pf(y)Ru and ~ Pf(y)Qu can be resolved to get RuQz. Note that the variable u in PxRu is not identified with the u of ~ Pf(y)Qu. At the beginning of the resolution operation variables are renamed so no variable is named in both clauses. The substitutions are always made uniformly at all occurrences of a variable in any clause, of course.

Let $\mathcal{A} = A_1A_2 \ldots A_n$ and $\mathcal{B} = B_1B_2 \ldots B_m$ represent two clauses, with A_i and B_k two literals, one from each clause. The resolvent of $\mathcal{A}$ and $\mathcal{B}$ with respect to A_i and B_k, if it exists, is found as follows.

(1) Check if A_i and B_k have the same predicate letter and precisely one has a negation sign. Halt, if this is not the case, since no resolvent clause with respect to A_i and B_k exists. Otherwise, proceed.

(2) Change variable names of $\mathcal{B}$ as necessary so that no variable name is shared by $\mathcal{A}$ and $\mathcal{B}$.

(3) Match $|A_i|$ and $|B_k|$, the atoms of A_i and B_k, respectively by appropriate substitutions in each literal (the matching procedure is outlined below). If the matching is unsuccessful, no resolvent exists, so halt. Otherwise, make the appropriate substitutions through $\mathcal{A}$ and $\mathcal{B}$ so that $\mathcal{A}'$ and $\mathcal{B}'$ are substitution instances of $\mathcal{A}$ and $\mathcal{B}$ and A_i and B_k are the complementary literals determined by the matching procedure.

(4) Form the resolvent by deleting A_i' and B_k' and putting the remaining literals of $\mathcal{A}'$ and $\mathcal{B}'$ minus redundancies in a single clause (as demonstrated above).

The matching procedure between atoms $|A_i|$ and $|B_k|$ finds a substitution instance for each atom to make them identical if such a substitution exists. The substitution obtained is in a suitable sense the most general substitution possible, an important property for attaining complete proof procedures. This can be performed as follows. Let $|A_i|$ and $B_k|$ share no variable names. The atoms are considered written as on paper in the notation adopted, all symbols in sequence.

(a) Set a pointed at the predicate letter of $|A_i|$ and a pointer at the predicate letter of $|B_k|$.

(b) Move the pointers in parallel to the right one symbol at a time (including parentheses if present). Stop at the first place the pointers point to different symbols. If the end is (simultaneously) reached first, the atoms match; exit. Otherwise, proceed to (c).

(c) If one pointer points to a parenthesis, or precisely one pointer has come to the end, there is a notational error (this shouldn't occur). If neither pointed points to a variable, match attempt fails; exit. If one pointer points to a function letter, search the function's arguments for an occurrence (at any depth) of the variable indicated by the other pointer. If an occurrence exists, match fails; exit. In all other cases, replace the variable v at one pointer by the the term whose first symbol is indicated by the other pointer. The replacement occurs at all occurrences of v in both literals. The two pointers now agree. Return to (b).

As an example, consider matching Pxf(y) and Pg(ww)w. A match exists, the common atom is Pg(f(y)f(y))f(y), whereas Pxf(y) and Pyf(g(x)) do not match. This latter example serves to emphasize that the variable renaming that is part of the resolution operation is not part of the matching operation.

One other necessary notion is that of a factor of a clause. A *factor* of clause $\mathcal{A}$ is a substitution instance $\mathcal{A}'$ of $\mathcal{A}$ determined by finding a match between two atoms of the clause that makes the two corresponding literals of $\mathcal{A}$ identical. Thus QaPf(a)a is a factor of QxPf(x)xPya. Note that here variables are not renamed to make each literal have distinct variables as substitutions are always uniform throughout clauses. Thus Pf(a)a is not a factor of Pf(x)xPxa. A factor of a factor of $\mathcal{A}$ is defined to be a factor of $\mathcal{A}$. Clearly a clause has only a finite number of factors. Example 1 below shows factoring is necessary for completeness. No refutation (see below) exists without factoring.

The *basic resolution* procedure proceeds as follows. Start with the set of given clauses as the "present" set.

> Form all possible resolvents of clauses of the present set and their factors. If the empty clause □ (the clause with not literals, the resolvent of two one-literal parent clauses) is formed, the given set represents an unsatisfiable formula. Otherwise, delete all tautologies (clauses with complementary literals). Define the remaining clauses as the present set and return to the beginning of this paragraph.

Each cycle of these instructions generates a new *level* of resolvents. The level of a clause is determined by the level of its first appearance. The calculation of all resolvents and factors of a given level at once is called a *level saturation* search.

The need for factors is seen from applying the resolution operation alone to the two member unsatisfiable set S of clauses given by S = {PuPv, ~ Px ~ Py}. Although clearly unsatisfiable (replace u,v,x,y by a) the resolvents are all two literal clauses. However, □ is derived in the first cycle of the basic resolution procedure, thus □ "occurs at level 1". See Example 1 below.

The justification for the correctness *(soundness)* and sufficiency *(completeness)* of the above procedure and those that follow come from the theorem due to Herbrand: If *S is a formula in Skolem functional form then S is unsatisfiable if and only if there exist clause substitution instances* $C_1, \ldots, C_n$ *such that* C_1 & C_2 & . . . & C_n *is unsatisfiable.*

Three examples follow of deductions of □, two of which are from examples of section 1. A *resolution deduction of* clause *C* is a sequence of clauses, each a (substitution) instance of a clause of the given set S, a factor of a preceding clause, or a resolvent of two preceding clauses of the deduction and with last clause C. A deduction of □ is called a *refutation.*

Example 1: A refutation of {PuPv, ~ Px ~ Py}:

1.	PuPv	given
2.	~ Px ~ Py	given
3.	Pu	factor of 1
4.	~ Px	factor of 2
5.	□	resolvent of 3,4

Example 2: A refutation of (11) of section 1 which in Skolem functional form appears as clauses (17), (18), (19):

1.	~ Ax Bx	(17)
2.	Aa	(18)
3.	~ Ba	(19)
4.	Ba	resolvent of 1,2
5.	□	resolvent of 3,4

Example 3: A refutation of the set of clauses (20) — (25), section 1:

1.	Pxyf(xy)	(20)
2.	~ Pxyu ~ Pyzv ~ PxvwPuzw	(21)
3.	~ Pxyu ~ Pyzv ~ PuzwPxvw	(22)
4.	Peyy	(23)
5.	Pg(y)ye	(24)
6.	~ Pag(a)e	(25)
7.	~ Pxya ~ Pyg(a)v ~ Pxve	resolvent of 6,2
8.	~ Pg(v)ya ~ Pyg(a)v	resolvent of 7,5
9.	~ Pg(g(a))ea	resolvent of 8,4
10.	~ Pg(g(a))yu ~ Pyze ~ Puza	resolvent of 9,3
11.	~ Pg(g(a))ye ~ Pyae	resolvent of 10,4
12.	~ Pg(g(a))g(a)e	resolvent of 11,5
13.	□	resolvent of 12,5

The reader can quickly check that axioms (i) through (xiii) together with the negation of the assertion for the monkey-banana problem define a set of clauses which yields a refutation.

3. The Unit Preference and Set-of-Support Strategies

Perhaps the earliest strategy to be applied to resolution is the *unit preference* strategy. The basic idea is simple: resolve with one-literal (i.e., unit) clauses more often than called for by the basic resolution procedure. One can indeed demand that the only resolution operations allowed will be when at least one clause is a unit clause. This may prevent finding, even theoretically, some refutations (i.e., this is not a *complete* process) but many sets of clauses can be shown unsatisfiable in this way with a great gain in search time over the basic procedure. Completeness is regained if the process intermittently forms the resolvents of various pairs of non-unit clauses so that eventually each level of resolvents is completed. Examples 1, 2, and 3 of the preceding section are "unit" proofs, i.e., have no resolvents of two non-unit clauses.

The *set-of-support* strategy sharply limits the clauses obtained at the first level by the basic resolution procedure. Let S denote the given set of clauses (e.g., lines 1-6, example 3) and T be any satisfiable subset of S (e.g., lines 1-5, example 3). The set-of-support strategy states that two clauses of T are *not* to be resolved together. A deduction under such constratint is said to "have set-of-support S-T". Example 3 is a refutation with set-of-support S-T where S-T contains only clause 6. The set-of-support strategy is complete. It is often used in conjunction with the unit preference strategy.

A very useful rule is the *subsumption principle*. Discard any clause C if there exists a clause D with a substitution instance D′ such that every literal of D′ is a literal of C. For example, Px *subsumes* PaQb so the latter should be discarded in the presence of Px.

Another rule of some usefulness, the *purity rule*, states that any clause may be discarded if it contains a literal with no complement among the substitution instances of the given set of clauses. Both rules are applicable to the combined unit preference, set-of-support strategy.

4. Partition Strategies

Let S be a given set of clauses. Let M be a set of literals such that no two literals have substitution instances which are complementary. We call M a *setting*. Thus {Px, ∼ Rf(y)} is a setting but {Px, ∼ Pf(y)} is not. A setting is a *partition* M_s of S if every literal of (a clause of) S is identical to, or a complement of, a substitution instance of a literal of M_s. Examples of partitions of S are (1) the set of all atoms of S and (2) the set of negations of all atoms of S. A *partition strategy* is a resolution procedure where one parent clause of every resolution operation contains *no* literal which is a substitution instance of a literal of M_s. Such a clause is called *false in* M_s. Example 3 of section 2 is a refutation realizable within the partition strategy where M_s is the set of all atoms of the given set S. Therefore, one parent clause of each resolution operation contains only negated atoms in this example. Partition strategies are complete for any choice of partition. Unit preference can be combined with partition strategies, often to advantage, with completeness maintained. A set-of-support strategy generally does not combine with a partition strategy to maintain completeness. However, a partition strategy can be regarded as a generalized set-of-support itself.

Ordering strategies can be superimposed on a partition strategy. This is highly effective in some cases and preserves completeness. For example, if the set S contains many predicate letters (such as occurs in information retrieval systems), assign an order to the predicate letters. Order all clauses that are false in M_s by ordering on predicate letters with the symbol $<$ (less than), also read "to the left of;" in every other clause form the maximal length subclause false in M_s ordered by the predicate letter ordering. Place any remaining literals of the clause after (to the right of) the ordered subclause; these literals may be placed in any desired order among themselves. If any clause false in M_s has several literals each of which could be rightmost, the clause is listed once for each of these "equal" literals with that literal occurring farthest right. A resolution operation is performed only if the rightmost literal of each clause is the literal resolved upon (also, of course, only if one parent is false in M_s). The resolvent is ordered as above. An example of a refutation realizable within this combined strategy is given below.

Example 4: The given set S of clauses is listed in lines 1, 3-6. M_s is the set of all negations of atoms of S. Note that line 2 gives the same clause as line 1 (clause 1 is false in M_s). The ordering of predicate letters is $A < B$.

1.	Aa Ab	given clause
2.	Ab Aa	clause 1, alternate order
3.	Ax Bx ~ Aa	given clause
4.	Ax ~ Aa ~ Bx	given clause
5.	Aa ~ Ax	given clause
6.	~ Aa ~ Ab	given clause
7.	Aa	resolvent of 1,5
8.	Ax Bx	resolvent of 7,3
9.	Ax ~ Aa	resolvent of 8,4
10.	Ax	resolvent of 9,7
11.	~ Aa	resolvent of 10,6
12.	□	resolvent of 11,7

5. Linear Resolution Strategies

The strategies considered in this section differ essentially from the preceding strategies which were oriented towards modifying, but not drastically altering, the level saturation search of basic resolution. Linear resolution is depth oriented with little interaction between clauses at a given level. The definition is best given in terms of a deduction (defined at the end of section 2).

A *linear* (resolution) *deduction* $\mathcal{D}$ *of clause* B_n is a resolution deduction $B_1, \ldots, B_n$ where B_i, $1 \leq i \leq k$ is a member, or a factor of a member, of the given set S and B_{k+i}, $1 \leq i \leq n-k$, is either a resolvent with B_{k+i-1} as one parent clause, called the *near* parent, or (B_{k+i} is) a factor of B_{k+i-1}. If B_{k+i} is a resolvent then the *far* parent of B_{k+i} must must be a B_m, or a factor of B_m, for some $m<k+i$. The initial sequence $B_1, \ldots, B_k$ of $\mathcal{D}$ is called the *prefix* of $\mathcal{D}$. Although the definition of linear deduction just given allows a far parent to be a factor not explicitly appearing in the deduction, it can be shown that completeness is preserved and little flexibility lost if such an implicit factoring is disallowed. A *linear refutation* is a linear deduction of □. Example 3 (section 2) is a linear refutation and Example 2 would be if lines 1 and 3 were interchanged. It might be instructive for the reader to find a linear refutation for the given set of Example 1.

A more restrictive complete resolution strategy is that of s-linear resolution (so-called because the added restriction can be expressed in terms of a subsumption condition). An *s-linear refutation* $\mathcal{D}$ is a linear refutation with the following restriction: the far parent of a resolvent is either chosen from the prefix of $\mathcal{D}$ or is chosen from $\mathcal{D}$ so that a certain *modified resolvent* is formed. The *modified resolvent* must be a factor of the usual resolvent and also a substitution instance of the near parent clause minus its literal resolved upon. For example, if the near parent clause is AxByCz, then the clause Aa ~ Cb is an acceptable far parent clause even if not in the prefix of $\mathcal{D}$ for then take AaBy, a factor of resolvent AxByAa, as the modified resolvent. If a modified resolvent exists, in general it can be found by factoring the resolvent on the literals that arise from the far parent, as done above.

The process of finding modified resolvents is more complex than finding standard resolvents, but fewer far parents are now acceptable which prunes the search space of deductions at any given depth of search. Example 5 below is an s-linear refutation where lines 8 and 11 satisfy the modified resolvent condition. (Line 8 is also permitted because the far parent is in the prefix of the deduction.)

A strategy called *merging* may be superimposed on s-linear refutation. A resolvent of two clauses, neither a tautology, is called a *merge* resolvent with *merge literals* $L_1, \ldots, L_n$, $n \geq 1$, if and only if the substitution instances of the two parents which form the resolvent propositionally each have $L_1, \ldots, L_n$ as literals. Thus AaBf(a) is a merge resolvent of AaByAg(x) and AxBf(x) ~ Ay. Note that the ordinary resolvent is AaByAxBf(x), so here a further substitution is necessary to produce a merge resolvent with merge literals Aa and Bf(a). The merge condition is added to s-linear resolution in the following way. If the far parent is not a member of the prefix of $\mathcal{D}$, it must be a merge resolvent with a merge literal as the literal resolved upon (in addition to the s-linear resolution requirement).

As with the strategies of the preceding sections, tautologies need not be used. However, there are some sets S for which there is a clause C such that if C is the first near parent clause, then there exists a refutation of S "from C" if and only if tautologies are permitted. However, a "set-of-support" condition holds. If S− {C} is satisfiable and S is unsatisfiable, then there exists a refutation from clause C in any of the strategies considered in this section with no tautology appearing.

There are further conditions which can be imposed on s-linear resolution. A deduction is *tight* if and only if no clause subsumes (see section 3 a later clause in the deduction. An s-linear deduction is an *ordered clause* deduction if all clauses are ordered (say, left to right when written) and every literal resolved on for near parent clauses and every literal factored is the rightmost literal of that clause. This last definition omits some technicalities (and improvements) which do not seriously alter the nature of the strategy. A version of merging can be incorporated with these strategies so that completeness is retained.

Example 3 gives a tight ordered clause deduction with line 6 as first near parent clause. The prefix is lines 1-6. Notice all far parents are members of the prefix. Such a refutation is called an *input* refutation. One of the more interesting results of resolution theory states that S has an input refutation if and only if it has a unit clause refutation. In general they are not the same refutation.

We give as Example 5 a tight ordered clause refutation with the merging condition that is not an input refutation.

Example 5: The given set of clauses Is that of Example 4 and appears on lines 1-5. Lines 1-5 needn't be considered ordered clauses as there is no order condition on the far parent. Line 5 is considered ordered when viewed as first near parent, however.

1.	Aa Ab		given
2.	~ Aa Ax Bx		given
3.	~ Bx Aa Ax		given
4.	~ Ax Aa		given
5.	~ Aa ~ Ab		given
6.	~ Aa Bb		resolvent of 5,2
7.	~ Aa Ab		resolvent of 6,3
8.	~ Aa	merge literal,	resolvent of 7,5
9.	Ab		resolvent of 8,1
10.	Aa		resolvent of 9,4
11.	□		resolvent of 10, line 8

The far parent condition of s-linear resolution and the tightness conditions, though strong, are expensive to check because in general several matchings need be checked per candidate. This costly multiple matching can be reduced to single matching by altering the resolution format of a tight ordered clause strategy in the following manner. When performing a resolution operation instead of deleting the rightmost literal of the near parent clause, retain it as a distinguished literal (we shall indicate such literals with **boldface**) and add the far parent minus the literal resolved upon to the right as before. Any literal identical to a near parent literal is not added. A new optional operation (reduction) is added also. If a distinguished literal can be made complementary to the rightmost literal in the clause, one forms the (ordered) clause with the appropriate substitution but with the rightmost literal deleted. Finally, one *always* deletes distinguished literals to the right of the rightmost ordinary literal as soon as that condition occurs within any operation. Example 6 gives the same refutation as Example 5 in the new notation.

Example 6:

1.	Aa Ab	given
2.	Ax Bx ~ Aa	given
3.	Ax ~ Aa ~ Bx	given
4.	Aa ~ Ax	given
5.	~ Aa ~ Ab	given
6.	~ Aa ~ **Ab** Bb	far parent, 2
7.	~ Aa ~ **Ab Bb** Ab	far parent, 3
8.	~ Aa	reduction
9.	~ **Aa** Ab	far parent, 1
10.	~ **Aa Ab** Aa	far parent, 4
11.	□	

This strategy actually is a special type of *model elimination* procedure developed independently of resolution theory. There is more to that procedure than outlined here, including some tests for rejection of bad deductions, use of "lemmas," etc. This can all be regarded as a refinement of linear resolution.

6. Implementation

Some variation of each of the strategies mentioned here has been realized on the computer. To the author's knowledge, no strategy of any of the three categories (sections 3, 4, and 5) has uniformly dominated the others. For example, the s-linear resolution strategy (actually in model elimination format) has given some dramatic results in search speed in cases where refutation existed by chance near the beginning of its search space (actually not too infrequent an occurrence because of multiple proof paths) but it can be equally unlucky and get "lost" in a search of outwardly, the "same size." The modified level saturation types seem to be more uniform in their performance, not reaching either extreme.

Completeness is not the final criterion of desirability for implementation. It is useful to know that when one piles the merging condition on top of tightness, plus ordering, on top of s-linearity on top of linearity that the result is a complete procedure, for at first glance it might seem unlikely to yield a single refutation. However, performance is the final arbiter. With this mind, programs exist that allow the superposition of strategies from two or even all these categories and some mixes have been quite beneficial for some sample problems. Also, "classical" heuristic procedures can be superimposed. One can envision a GPS-type difference analysis coupled to a linear resolution strategy, for example.

Resolution procedures have proved simple elementary (i.e., formal) group theory and elementary number theory problems (the latter when appropriate lemmas were given in the axiom set). Although these are real mathematical problems, and beyond the theorem provers of a decade ago, they are homework exercises for mathematics majors. Resolution procedures occur in some question-answerer (q-a) systems as the inference device. Such systems apparently are among the most competent q-a systems in existence. A system using resolution strategies has verified recently published results on Boolean algebras whose proofs were not totally trivial.

7. Credits

The intent in this section is not to name all the people who have contributed to aspects of resolution theory summarized above but to name some who have their names associated with specific items highlighted here. J. A. Robinson (about 1965) introduced basic resolution; also P_1 deduction and Hyper-resolution, which were the earliest and most basic partition strategies. Prawitz in 1960 outlined the notion of matching for proof procedures. Wos, G. Robinson and Carson introduced unit preference and set-of-support. Slagle, Luckham, and Meltzer independently worked with partition strategies. Slagle, and, independently, Kowalski and Hayes, superimposed the order strategy on partition resolution. The author and Luckham independently introduced linear resolution. S-linear resolution and model elimination are the author's. Merging is due to Andrews. Chang discovered the relationship between unit and prefix resolutions. Anderson, and independently Yates, Raphael, and Hart linked merging and s-linear resolution.

References

CMU Papers

1. Andrews, P., "Resolution with merging," *J.ACM 15* (July 1968), 367-381.
2. Andrews, P., "Resolution in type theory," CMU report, 70-27.
3. Loveland, D., "A simplified format for model elimination," *J.ACM 16* (July 1969), 349-363.
4. Loveland, D., "Theorem-provers combining model elimination and resolution," *Machine Intelligence 4*, (Ed. Meltzer and Michie) Edinburgh University Press, Edinburgh (1969), 73-68.
5. Loveland, D., "A linear format for resolution," *Lecture Notes in Mathematics 125*, Springer-Verlag, Berlin (1970), 147-162.
6. Loveland, D., "Some linear Herbrand proof procedures: an analysis," CMU report (Computer Science), December 1970.

General Papers

7. Allen, J. and Luckham, D., "An interactive theorem-proving program," *Machine Intelligence 5*, (Ed. Meltzer and Michie) American Elsevier Publishing Co., New York (1970), 321-336.
8. Anderson, R. and Bledsoe, W. W., "A linear format for resolution with merging and a new technique for establishing completeness," *J.ACM 17* (July 1970), 525-534.
9. Chang, C. L., "The unit proof and the input proof in theorem-proving," to appear in *J.ACM.*
10. Meltzer, B., "Power amplification for automatic theorem-provers," *Machine Intelligence* 5, Ibid.
11. Mendelson, E., *Introduction to Mathematical Logic*, Van Nostrand, Princeton, 1964, 300 + pp. (see chapters 1 and 2).
12. Robinson, J. A., "A machine-oriented logic based on the resolution principle," *J.ACM 12* (January 1965), 23-41.

[illegible]

Control Structures

David A. Fisher

Computer science is concerned with automata which can perform various operations. These operations differ from the usual functions of mathematics in that there is a time interval associated with each one. Consequently, the computer scientist is concerned with algorithms, i.e., with sequences of operations which approximate functions of logic in a finite time. Any single time ordered sequence of operations performed in an execution of an algorithm is called a *control path.* Those operations which determine the control path(s) to be followed through an algorithm are called *control operations.* The control operations of an algorithm together with the interpretation rules for the non-control operations constitute the *control structure* of the algorithm.

Despite the importance of control to computer science, until recently[1] little direct attention has been given to it. What are the control structures of current programming languages? Are there other useful control structures? How can control be described formally? Can control structures be composed, say, in the manner of functions? Can the use of control structures better suited to a task simplify that task or expose the significant problems of that task?

No attempt will be made to answer all of these questions here. Instead, a set of primitive control operations will be defined from which it is claimed all others can be formed by composition. A few specialized control structures will then be defined by composition of these primitives.

A set of primitive control operations should be large enough to span the space of control structures but small enough to be manageable. Care must be exercised with the first requirement. It might be concluded that sequential processing and a conditional control, the only control structures of the Turing machine, are sufficient. There are, however, useful and interesting properties of control structures that cannot be described in terms of just these two controls. If parallel processing were described by its simulation using sequential control primitives and by means of some scheduling algorithm, then, though one might come to understand parallel processing as a particular interleaved execution of sequential processes, the idea of concurrent execution would not be conveyed. Concurrency is a concept which cannot be composed purely from sequential primitives.

Functional notation (e.g., $f(x_1,x_2,\dots x_n)$ where $0 \leq n$ and the x_i's may themselves be functional forms) will be used to describe the primitive control operations and their compositions. For a non-control operation f, $f(x_1,x_2,\dots x_n)$ will be given the following interpretation: evaluate each of the x_i's and apply the operation f to the resulting values. This implies that each of the arguments to a non-control operation must produce a value.

For control operations the same notation will be used, but each operation will have a unique interpretation. For example, the control operation "*if*(c,x,y)" might be defined as follows: (1) Evaluate c (assumed to be a boolean expression). (2) If the resulting value is *true*, then evaluate x and take that resulting value as the value of *if*. (3) Otherwise evaluate y and take that resulting value as the value of *if*. In no case are both x and y to be evaluated. Note that to describe *if* as above the expressions for x and y rather than their values are treated as arguments to *if*. Thus the control operations are those operations which have expressions as their arguments. In the definitions below, those formal parameter names which refer to expressions are in **boldface.**

The controls to be proposed are primitive in the sense that each encompasses only a single idea. This contrasts with many of the specalized controls (e.g., *for* statements of Algol-60) of current languages. This makes it possible to provide accurate descriptions of the other control structures and to guarantee that composition will be meaningful, (e.g., with *for*, composition is restricted to strict imbedding). The proposed control operations follow.

The Primitive Operations

seq($\mathbf{x}_1$,$\mathbf{x}_2$, . . . $\mathbf{x}_n$) *Sequential processing.*

The sequential processing operation *seq* causes its arguments to be evaluated in the order given from left to right in such a way that the evaluation of an argument will not begin until the evaluation of all arguments to its left are complete. Only that value produced by the evaluation of x_n will be taken as the value of *seq*. This means that the purpose of the evaluations of $x_1,x_2, \ldots x_{n-1}$ must have been side effects (i.e., changes they impose on the environment). Evaluation for side effects only is usually called execution, although we will use the terms interchangeably.

par($\mathbf{x}_1$,$\mathbf{x}_2$, . . . $\mathbf{x}_n$) *Parallel processing.*

The parallel processing control operation *par* causes its arguments to be evaluated independently (i.e., in parallel) as if each had its own processor. No assumptions are made about the relative speeds of these processors, so there are no guarantees concerning the chronogolical order of their side effects. These effects can, however, be controlled by the synchronization of the parallel control paths as described below. The establishment of parallel paths has no special effect on the environmental data structure of a program. In particular, copies of the environment are not made for each path, so that they must share the same global variables and define their own local variables. The value of x_n is taken as the value of *par* but is returned only after the execution of all the x_i's is complete.

cond($\mathbf{c}_1$,$\mathbf{x}_1$,$\mathbf{c}_2$,$\mathbf{x}_2$, . . .) *Condition testing.*

The conditional operation *cond* has an even number of arguments. Beginning with the leftmost argument, every other argument will be evaluated in order from left to right until one having value *true* is encountered. The argument immediately to the right of this argument will then be evaluated and the result taken as the value of *cond*. If none of the odd numbered arguments evaluates to *true*, the value of the expression is *undefined* and none of the even numbered arguments is evaluated.

monitor($\mathbf{c}$,$\mathbf{x}$) *Monitoring.*

The monitor operation *monitor* causes the expression c to be continuously evaluated until its value becomes *true*. The value of c can change only when the value of at least one of the variables of the expression c is changed as a side effect of a parallel path. (In practice c might be reevaluated only when any of its variables changes.) When the value of c becomes *true*, the repetitive evaluation of c will be terminated, the expression x will be evaluated, and the resulting value taken as the value of *monitor*. No assumption is made about the relative speeds of the processors which change the variables of c and the processor which evaluates c. Thus, as in parallel processing, synchronization is handled by a separate control operation as described below.

synch(d,**x**,**y**) *Synchronizing.*

The operation *synch* provides a number of equivalent functions: synchronization of parallel or interleaved processes, mutual exclusion,[2] and indivisibility of operations. The argument d must be a data structure. The argument x is an expression which will be evaluated only if no other *synch* with d as its first argument is simultaneously evaluating its second argument. The argument y is an expression which will be evaluated whenever x cannot be evaluated. Thus, if several *synch* operations are executed simultaneously on the same datum d then exactly one will have its second argument evaluated (which one is, however, undetermined). In no case does the *synch* operation cause waiting as with *monitor.*

The *synch* operation can be used for mutual exclusion to guarantee that only one process at a time executes a critical section of a program or modifies a datum. *Synch* also provides for the relative indivisibility of operations because, for all *synch* operations on a given datum, the evaluation of their second arguments cannot be simultaneous or interleaved (i.e., they are indivisible with respect to that datum).

cont(**x**) *Continuous processing.*

One control path will be called continuous with respect to another control path if and only if all non-monitoring operations of the former path occur between two consecutive steps (i.e., state changes) of the latter. The control operation *cont* causes the evaluation of its argument to be continuous with respect to all other control paths. If two *cont* operations are executed simultaneously then the evaluation of both their arguments would be continuous with respect to other parallel control paths, but would be as normal control paths with respect to each other (e.g., *synch* would be required for mutual exclusion). Because continuous and parallel processing operations can be embedded within each other to arbitrary depth, many levels of relative continuity can occur. These roughly approximate the priority levels found in some operating systems.

return(v,p) *Call and return.*

The previous operations can be combined using functional composition to define new control operations (e.g., f). Whenever such an operation is called (e.g., $f(x_1,x_2,\dots)$) then the actual parameters, the x_i's, are associated with the corresponding formal parameter names and control is passed to the expression defining f. The object which describes the dynamic state of the evaluation for a single activation of an operation will be called a *process*. The calling operation is implicit in the use of functional notation and need not be symbolized separately.

In many cases control is to be returned to the calling process after the expression defining the operation has been evaluated. The control operation return(v,p) causes both control and value v to be returned to the process p (usually the caller).

self() and caller()

The operations *self and caller* although not control operations will be useful in describing other operations. The value of *self* is the process in which it is executed. The value of *caller* is the process which called the current process.

quote(**x**) and eval(**y**,e)

The *quote* operation is used to inhibit the evaluation of the expression x. *Eval* evaluates the expression y in the context of environment (i.e., process e. Thus for any expression x:

x= eval (quote(x),self()).

Examples of Control Structure Construction

Consider now some specific control structures which can be built from the preceding primitives. One commonly used control is the *coroutine.*[1] A coroutine relation between two processes is similar to the subroutine in that the initial call builds a new process and passes control to that process. The coroutine process, however, may pass control back to its caller before reaching its final return. Intermediate coroutine returns can be defined as follows:

```
cocall(v,p) = return(v,p).
```

Here v is the value to be returned and p is the coprocess (either caller or called). Notice the symmetry which permits each of the coprocesses to treat the other as if it were a subroutine (i.e., calling the process with the routine *cocall* and receiving the return value via the *return* in the expression defining *cocall.*

For the remaining examples additional mnemonic notation will be used. Infix operators will be used for noncontrol operations, "x" will be used for quote(x), f[x_1,x_2, . . .] will be used for f("x_1","x_2", . . .), *x will be used for return (x,caller()), {$c_1 \rightarrow x_1$; $c_2 \rightarrow x_2$; . . .} will be used for cond("c_1","x_1","c_2","x_2", . . .), and x·e will be used to reference the variable whose name is the value of x in the context of the environment which is the value of e.

Another useful control structure is iteration. This might take the form *while*(**c**,**x**,e) where the expression x is repetitively evaluated as long as the value of c is *true* and both c and x are to be evaluated in the context of environment e. To be more precise, *while* (*c*,*x*,e) first evaluates c in the context of e. If the resulting value is *true*, then first x is evaluated in the context of e and second (i.e., after the evaluation of x is completed) the *while* operation is repeated. If an evaluation of c does not yield *true* then control is returned to the calling process. Note the similarity between the above English definition of *while* (the previous three sentences) and the formal definition as a composition of primitive control operations below:

```
while(c,x,e) =* {eval(c,e) → seq[eval(x,e);
                                 while(c,x,e)]}.
```

The *while* operation can be used to define more specialized iterative control structures. The Algol-60 form *for* I=A *step* B *until* C *do* S has several interpretations [4]. A liberal interpretation of the Algol report [5] would allow the expressions, I, A, B,and C to be evaluated only once giving the following definition

```
for(I,A,B,C,S) =* seq[I·caller() := A; while("A-c*
                                    sign(B) ≦ 0",
     "seq[eval(S,caller()); I·caller() := A := A+B]",
                                          self() )].
```

The notation I·caller() is used to indicate the (unique) I in the environment of the caller existent at the call.

A strict interpretation, however, would require that all the arguments be evaluated for each iteration (in fact that B be evaluated twice and the address of I evaluated three times) as follows:

```
for(I,A,B,C,S) =* seq[eval(I,caller())·caller() :=
                                  eval(A,caller());
   while("eval(I,caller())·caller()
     -eval(C,caller())xsign(eval(B,caller())) ≦ 0",
     "seq[eval(S,caller()); eval(I,caller())·caller() :=
           eval(I,caller())·caller()+eval(B,caller())",
     self())].
```

The Algol condition *if* C *then* X *else* Y can be defined as:

```
if(C,X,Y) =* { C → eval(X,caller());
               true → eval(Y,caller())}.
```

A more unusual control structure is the *sidetrack* control[3] which can be used to describe breadth first tree search algorithms (e.g., recognizers for context-free grammars). The operation *sidetrack*(*x*,*y*) causes the expressions x and y to be evaluated in parallel. The expressions represent alternative branches of a binary search tree. When a failure condition is encountered in one of these branches the associated control path will be terminated by execution of the operation *terminate*(). If a branch is successful, control (and possibly a value) will be returned to the caller of *sidetrack*. Because both branches might be successful (e.g., an ambiguous string in the case of a parser), there may be several returns of control to the same process. To prevent the resulting conflict of states, all returns will be made using the multiple parallel return operation *mpr* which returns not to the specified process but to a copy of it. The operations *sidetrack*, and *mpr* are defined below:

```
    sidetrack(x,y,p) = par[mpr(x,p); mpr(y,p)]
    mpr(x,p) = return(eval(x,p),copy(p))
```

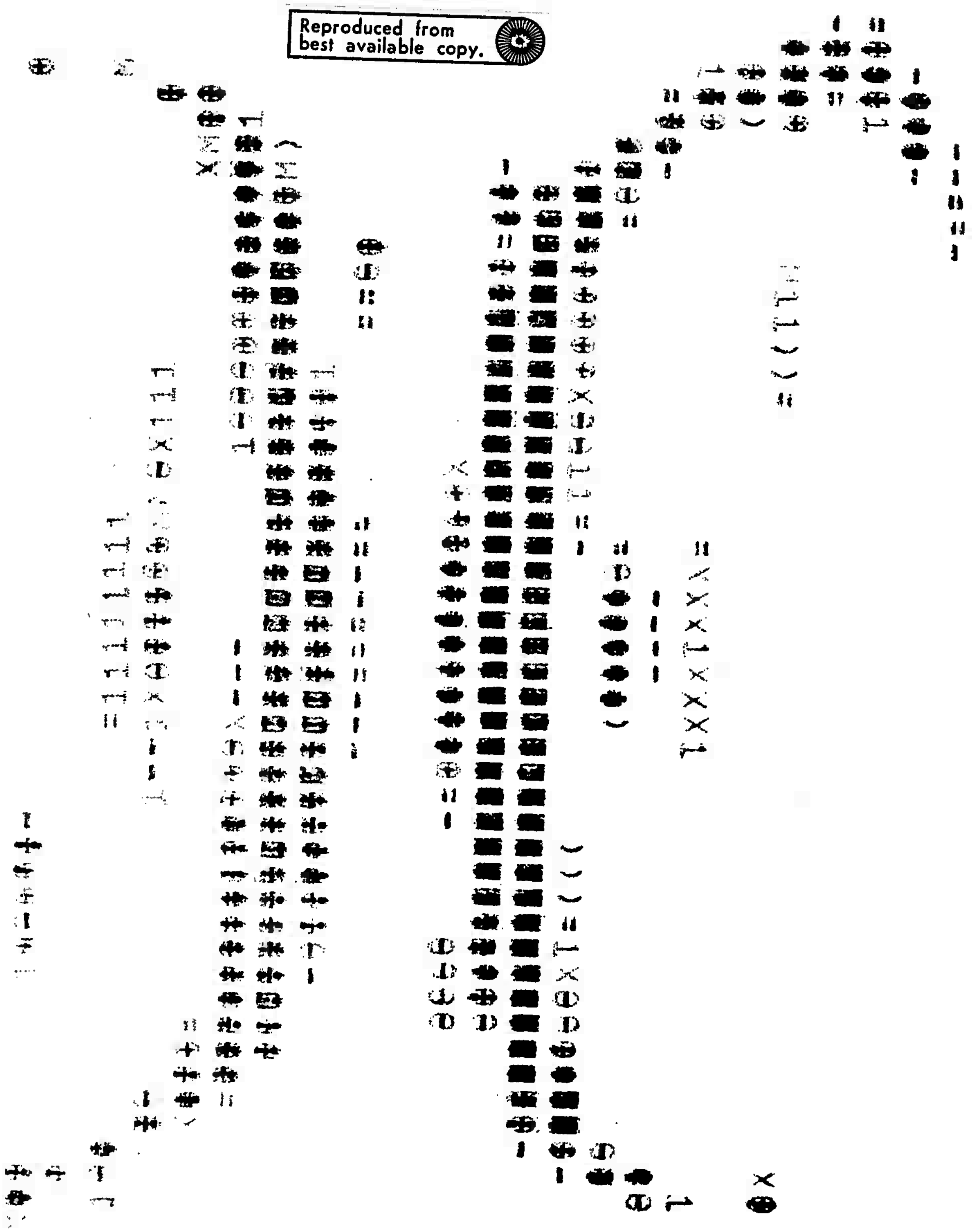
A recognizer R for terminal strings of r where r ::= A|B r could be written as:

```
    R(string) = sidetrack("{head(string) = 'A' →
                    tail(string); true → terminate()}",
        "{head(string)= 'B' → R(tail(string)); true →
                                terminate()}",self())
```

Understanding of control is in its infancy. The above has attempted to isolate and solidify some of the concepts and indicate the potential for formal means for defining control. Here the approach was to define control structures as compositions of a given set of primitive control operations. The choice of primitives was somewhat arbitrary within the requirements for simplicity of the individual primitives and that they span one's intuitive understanding of the space of control structures. Other choices could have been made (in fact, a slightly modified set of primitives is being implemented to satisfy an additional requirement: run time efficiency on existing hardware). Finally, a few examples have been givon to illustrate some of the variety in control structures and to demonstrate control definition by composition of the proposed primitives. Additional examples are given in reference 3.

References

1. Conway, M. E., "Design of a Separable Transition-Diagram Compiler," *Comm. ACM 6* (July 1963), p. 396-408.
2. Habermann, A. N., "On the Harmonious Cooperation of Abstract Machines," Doctoral Dissertation, Technische Hogeschool. Eindhoven, The Netherlands, October 1967.
3. Fisher, David A., "Control Structures for Programming Languages," Doctoral Dissertation, Carnegie-Mellon University, Pittsburgh, Pa., May 1970.
4. Knuth, Donald E., "The Remaining Trouble Spots in ALGOL 60," *Comm. ACM 10* (October 1967), p. 611-618.
5. Naur, P. (Ed.) "Revised Report on the Algorithmic Language ALGOL-60," *Comm. ACM 6* (January 1963), p. 1-17.



Bliss: A Language for Programming Systems

William A. Wulf

The development of sophisticated *programming systems,* notably programming languages and operating systems, has largely been responsible for the increasingly wide application of computers. The primary objective of these systems is to permit the solution of a problem to be stated more concisely, and in terms more natural to the problem, than is possible with the numeric instruction encoding interpreted by computer hardware. This paper deals with a programming system, specifically a programming language, which is designed primarily for writing other programming systems.

It is curious, but true, that although great strides have been made in creating programming systems to support the programming activities of application areas, systems programmers themselves have been the beneficiary of almost none of this progress. In particular, the vast majority of programming systems continue to be written in assembly language—scarcely one step removed from the hardware numeric encodings. Why? Why haven't systems programmers chosen to write their systems using existing programming systems, particularly a programming language, and why hasn't their management insisted upon it? The advantage of using a so-called "higher-level" language are well known and thoroughly documented: programmers are most productive, programs contain fewer errors and are more easily repaired, programs are more easily understood and modified, etc.

Part of the reason why systems programmers continue to use archaic tools is simply inertia—it's always been done that way. A more significant reason, however, is the feeling on the part of practicing systems programmers that existing programming languages are not appropriate for the kind of work that they do. In this context the question of why systems programmers don't use a higher-level language becomes: "What about systems programming is different from other programming tasks, and how should these differences manifest themselves in a programming language specifically designed for this application?"

In many ways systems programming is like other programming applications. Thus, for example, algorithms used in the construction of assemblers, compilers, interpreters, operating systems, etc., are certainly different from those which control missle trajectories, simulate the behavior of a nuclear reactor core, or produce a corporate payroll. But many of the issues which arise in the implementation of these algorithms are similar. Issues such as the order in which numeric computations are performed in order to maintain accuracy are as important in a compiler's number conversion routines as in reactor simulations; proper overlapping of input-output with computation is critical to systems programs as well as to payroll programs; etc. There are differences in emphasis between these applications, however, which give rise to differences in language features. Some of the more Important differences for systems programs are:

— efficiency
— access to all hardware features
— minimal run-time support
— evolution of the resultant system.

Ideally all programs would be maximally efficient—rapid in execution and conservative in their use of storage. In many applications, however, economy in programming effort and conciseness in expression are traded for execution speed and use of storage. No programmer deliberately writes a program to be slow or wasteful of storage; however, due to the high frequency of the use of systems programs the emphasis on efficiency is generally greater than in other applications.

Most programming systems hide from their user, for his own protection and convenience, idiosyncracies of particular hardware machines. To the system programmer, however, these idiosyncracies are an integral part of the problem and must not be hidden.

Many features of modern programming systems are not available directly in the hardware on which they run. Instead, these features are implemented with software "run-time support" programs—of whose existence the user need not be consciously aware. A simple example is the trigonometric "sine" function in most scientific programming languages. Few computers have this function in hardware—rather it is evaluated by a subprogram automatically included in the user's program by the programming system without any overt action on the user's part. The systems programmer, whose task it is to create such support, cannot in turn require it. The systems programmer must be able to create and exploit his own support, thus "bootstrapping" to increasingly sophisticated systems.

A final observation on programming systems is, perhaps, the most important of all. Programming systems are never finished but are in a constant state of evolution. New features are constantly added and old errors repaired. The more heavily a system is used, the more rapid the rate of evolution and repair. This situation seems inevitable so long as new application areas, all with slightly differing requirements, continue to emerge.

A central problem of devising a language for systems programming would appear to be that of providing mechanisms for enabling the programmer to cope with this evolution (of programs written in the language) while satisfying the other three criteria mentioned earlier: efficiency, access to the hardware, and minimal run-time support.

The present state in programming systems relative to coping with evolution is summed up in the programming jargon word: "kludge".

"I just had a neat idea for a new feature. Now, the system wasn't meant to do this, *but* . . . if I contort this a little, and rebuild that, and since no one uses these bits . . . Gee, I think it'll work."

That's a kludge with a small "k". Now, repeat the process fifty, or a hundred, or a thousand times by many different people at computer installations scattered across the country and the system becomes a Kludge. Unfortunately almost all existing systems are Kludges. The property of systems which results in their evolution toward Kludges, and which a systems programming language must correct, is the ease in making trivial changes and the difficulty in making fundamental changes to systems. The consequences of this property is the introduction of peripheral modifications which subvert and distort the original structure of the system and lead ultimately to inefficient, "dirty" systems.

The remainder of this paper is devoted to the description of a language, Bliss, which is designed to satisfy the goals set out above for a systems programming language.

The Representation of Data Structures: A Thesis

The surface structure of the Bliss programming language is a logical, but not especially innovative, evolution from the Algol-60 family of languages. There is a central aspect of the language, however, which distinguishes it from other members of this family, that of the representation of data structures. That role of representation follows from the thesis that: "The central issue in systems programming is that of the representation of data structures. This issue is the key to both efficiency and to the rational evolution of programming systems." Subsequent sections will deal with the structure of the Bliss language and in particular with the manifestation of the concern with representation expressed by the thesis. In this section we are concerned only with the meaning and implications of the thesis.

All programming deals with structured information—that is, with atomic information items which not only have a value, but also bear some relation to other atomic information items. These relationships are expressed in programming languages by "data structures" (such as arrays, lists, queues, stacks, etc.) which are used to model the real relations which exist between data items. Most programming languages contain a fixed set of such data structures; namely, those deemed appropriate to an application area by the designers of the language. Scientific languages such as Algol and Fortran, for example, contain only arrays (as models of the mathematical vector and matrix structures) while string processing languages such as SNOBOL contain sequences of characters as intrinsic structures. These data structures are, in turn, represented, or modeled, by the implementor of the programming system in terms of the explicit data structures of a hardware computer.

So long as the data structures required by an application area are small in number and fairly uniform over the area, as is largely the case in scientific applications, this situation is acceptable. In systems programming, however, this is decidedly not the case. All of the structures mentioned above are used, as well as many others that have no generic name. Furthermore, in the interest of efficiency, many different representations of the same logical structure are used. To achieve reasonable efficiency it is imperative that a language to be used for systems programming permit the definition of the representations to be used.

When a programming system is initially built, the conscientious systems programmer devises representations to maximize the efficiency of his system. As new features are added to a system and it evolves, the original representations may no longer be the most appropriate. To the extent to which new, more natural representations cannot be provided, the original representations will be modified, thus circumventing the reorganization that evolution requires, ergo a kludge.

In order to achieve the criteria outlined above, two principles were followed in the design of the data structure facility of Bliss:

— The user must be able to specify the *accessing algorithm* for elements of a structure (which is equivalent to specifying the representation) at as low a level as he deems necessary.

— The structure definition and the algorithms which operate on the elements of a structure must be separated in such a way that either can be modified without affecting the other.

It is customary in conventional programming languages to include a number of implicit data structures (such as arrays, lists, strings, and so forth) as an integral part of the language. The implementor of these languages chooses a representation for these structures and the languages user has no control over them. Bliss takes an entirely different approach. The first of these principles reflects the need for flexibility and efficiency in systems programming. It also reflects the conviction that the designer of Bliss, cannot—and must not—predict which representations a system programmer will need. Any given set of primitive structures, with or without the ability to define more complex structures from them, would be totally inappropriate. Instead, representations of structures are defined in Bliss in terms of the computational procedure (the algorithm) by which elements of that structure are accessed. Thus no decisions are made *a priori* by the Bliss implementation concerning the most appropriate representation of even the most elementary structures, and the user has maximal flexibility in choosing one.

The second principle reflects a concern for the modification of systems written in Bliss. So long as the representation of structures is separated from the algorithms which operate on data contained in those structures, and the representations are easily modifiable, it is possible for systems to evolve in an orderly fashion. Of course, providing such a facility does not guarantee that it will be used intelligently—but that is an educational, and possibly managerial, issue and not a technical one.

Most existing languages satisfy one or the other of the above criteria but not both. Assembly languages allow (indeed, demand) algorithmic specification of data structures since they have no implicit data structures other than those representable in the hardware itself. The specification of a structure in an assembly language program need not be localized, however, and is usually distributed to all places where an element of the structure is accessed, thus making it difficult to modify. Algol, Fortran, etc., on the other hand, localize the specification of a structure to the place where it is declared but do not allow its representation to be defined by the programmer.

Description of Bliss

Bliss may be characterized as an Algol derivative in the sense that it has a similar expression format and operator hierarchy, a block structure with lexically and dynamically local variables, similar conditional and interative constructions, and (potentially) recursive procedures. The similarity stops shortly beyond this surface comparison, however. Bliss will be described in terms of its major aspects: (1) the underlying storage concept, (2) control, and (3) data structures. A complete definition of the language may be found in the Bliss Reference Manual.[1]

1. Storage Names and Identifiers

In order to implement the objectives set out previously concerning the representation of data structures it is first necessary to adopt precise and consistent interpretations of the concepts of "identifier", "name", and "value". The distinction between these concepts is at best fuzzy in most programming languages. The distinction to be made is that between an object, a name for the object, and a name of the name. In English prose if one wishes to talk about the smallest integer larger than five, one writes:

6

The above mark serves as a name for that particular integer, but there are many others; for instance,

Six 卌 I VI 5+1

Now, if one wishes to talk about one of these specific names of six, one encloses it in quotes; thus "6" serves as the name of a particular graphic character which, in turn, names a specific integer.

Bliss makes a similar distinction except that the objects named are *variables*, i.e., their value may change in time, and are represented by some storage media in the computer. The name of a variable, also called a *pointer* to the variable is encoded as a bit pattern which itself may be manipulated and in particular may be the value of another named object. An identifier serves as the name of a name of (pointer to) a variable. To complete our analogy, then, a Bliss identifier serves the role of the quoting device in English, pointers correspond to simple names, and the objects ultimately named are variables represented in storage. The weakness in this analogy is that English provides few mechanisms for performing operations on names while in Bliss names are encoded as bit patterns and hence may be operated upon by any operator in the language (see below).

A Bliss program operates with and on a number of storage *segments*. A storage segment consists of a fixed and finite number of *words*, each of which is composed of a fixed and finite number of *bits*. A contiguous set of bits within a word is called a *field*. A field may be *named*, the value of a name is also called a *pointer* to the field. In particular, an entire word is a field and may be named.

In practice a segment generally contains either program or data, and if the latter, it generally is an integer number, a floating point number, character(s), or a pointer to other data. To a Bliss program, however, a field merely contains a pattern of bits on which the programmer may place any interpretation he chooses. Various specific operations are defined in Bliss and may be applied to fields and bit patterns, such as: fetching a bit patterns (value) from a field, storing a bit pattern into a field, arithmetic, comparison, and boolean operations on bit patterns, and so on. These operations are roughly those provided by the hardware. From these all other programmer-defined operations must be built. The interpretation placed upon a bit pattern and consequent transformation performed by an operator is an intrinsic property of the *operator*, and not of its operands. In particular, names (pointers) are bit patterns and as such are manipulable objects in the language.

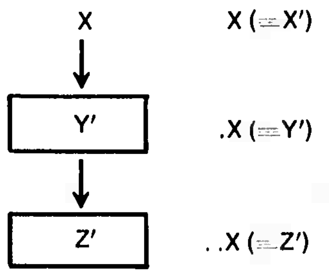
Segments and identifiers are introduced into a Bliss program by declarations, called 'allocation declarations'; for example:

```
global g;
own x,y[5], z;
local p[100];
```

Each declaration introduces one or more segments and binds the identifiers mentioned to the name of the first word of the associated segment.

The segments introduced by declarations contain one or more words; the size of a segment may be specified (as in "*local* p[100]") or defauted to one (as in "*global* g;"). The identifiers introduced by a declaration are lexically local to the block in which the declaration is made (that is, they obey the usual Algol scope rules) with one exception—namely, *global* identifiers are available to other, separately compiled programs. Segments created by *own* and *global* declarations are created only once and are preserved for the duration of the execution of a program. Segments created by *local* declarations are created at the time of block entry and are preserved only for the duration of the execution of that block. Reentry of a block before it is exited (by recursive function calls, for example) behaves such that *local* segments are dynamically local to each incarnation of the block.

It is important to reiterate that identifiers are bound to names by declarations, and that a name is a pointer (a particular interpretation on a bit pattern). Thus, the value of an instance of an identifier, say x, is a name of, or pointer to, x, *not* the value of the field named by x. Moreover, operators may be applied to names to yield new names. This interpretation requires a "contents of" or "value of" operator for which the symbol "." has been chosen. The operator "." may be applied to any expression—thus placing a "pointer" interpretation on the bit pattern which results from evaluating that expression. Thus the value of the expression "X" is the name (a pointer) of a specific variable, X, ".X" is the bit pattern stored in X and "..X" is the value of a variable whose name is stored in X. If we denote the bit pattern which, when interpreted as a pointer, names X by X′ and use boxes to represent storage cells, then the situation described above is:



There are two additional declarations whose effect is to bind identifiers to values (possibly names), but which do not create segments; examples are:

external s;

bind y2 = y+2, pa = p+.a;

An *external* declaration binds one or more identifiers to the names represented by the same identifier declared *global* in another program. The *bind* declaration binds one or more identifiers to the value of an expression at block entry. Potentially the value of this expression may not be calculable until run-time, e.g., as in "pa=p+.a" above.

2. Operations and Control

Bliss is an *expression language;* that is, every executable construct, including those which manifest control, is an expression and computes a value. There are no statements in the sense of Algol or PL/I. Expressions may be concatenated with semicolons to form compound expressions, where the value of a compound expression is that of its last (rightmost) component expression. Thus ";" may be thought of as a dyadic operator whose value is simply that of its righthand operand. A pair of symbols *begin* and *end*, or left and right parentheses, may be used to embrace such a compound expression and convert it into a simple expression. A block is merely a special case of this construction which happens to contain declarations; thus the value of a block is defined to be the value of its constituent expression.

The assignment operation, "←", is a dyadic operator whose left operand is interpreted as a pointer and whose right operand is an uninterpreted bit pattern. The right operand is stored into the field named by the left operand; the value of the expression is that of its right operand. Recalling the interpretation of identifiers and the "." operator, the expression

x ← .x+1

causes the value of the field named by x to be incremented by one. The value of the entire assignment expression is that of the incremented value.

There are five forms of explicit control expression: conditional, loop, case-select, function, and escape.

The conditional expression

if ϵ_1 *then* ϵ_2 *else* ϵ_3

is defined to have the value of the expression ϵ_2 just in the case that the rightmost bit of expression ϵ_1 is a 1; it has the value of ϵ_3 otherwise. The abbreviated form "*if* ϵ_1 *then* ϵ_2" is considered to be identical to "*if* ϵ_1 *then* ϵ_2 *else* 0".

Whereas the conditional expression provides two-way branching, the *case** and *select** expression provide more general n-way branching:

case e *of set* ϵ_0; ϵ_1; . . .; ϵ_{n-1}; ϵ_n *tes*
select e *of nset* ϵ_0: ϵ_1; ϵ_2: ϵ_3; . . .;
ϵ_{2n}: ϵ_{2n+1} *tesn*

The value of a *case* expression is ϵ_e; that is, the expression e is evaluated and this value is used as an index to select one of the expressions ϵ_i ($0 \leq i \leq n$), which then becomes the value of the entire *case* expression.

The *select* expression is somewhat similar to the *case* expression except that the expression e is not used as a simple index, and hence not restricted to the range $0 \leq e \leq n$. Instead, after e has been evaluated its value is successively compared with the first element of each of the pairs ϵ_{2i}: ϵ_{2i+1} in the order of increasing values of i. For each pair such that $e = \epsilon_{2i}$ the the second element of the pair, ϵ_{2i+1}, is also executed and the last of these to be executed defines the value of the entire *select* expression.

Loop expressions imply repeated execution (possibly zero times) of an expression until a specific condition is satisfied. There are several forms, of which we shall mention three:

while ϵ_1 *do* ϵ
do ϵ *while* ϵ_1
incr <id> *from* ϵ_1 *to* ϵ_2 *by* ϵ_3 *do* ϵ

In the first form the expression ϵ is repeated so long as the rightmost bit of ϵ_1 remains 1. The second form is similar except that ϵ is evaluated before ϵ_1, thus guaranteeing at least one execution of ϵ. The last form is similar to the familiar "*step . . . until*" construct of Algol, except (1) the control variable, <id>, is local to ϵ, and (2) ϵ_1, ϵ_2 and ϵ_3 are computed only once (before entry to the loop). Except for the possibility of an escape expression within ϵ (see below) the value of a loop expression is uniformly taken to be -1.

Invocation of functions (subroutines) is specified by the usual notation:

$$\epsilon(\epsilon_1, \epsilon_2 \ldots \epsilon_n)$$

This expression causes activation of the segment named by ϵ passing the values $\epsilon_1, \ldots, \epsilon_n$ as parameters. The value of a function call is that resulting from execution of the body of the named function.

The familiar "goto . . . label" form of control has not been included in Bliss. Unrestricted goto's require considerable run-time support (principally due to the possibility of jumping out of functions and/or blocks). More importantly, the use of the general goto, because of the implied violation of program structure, is a major villain in making programs difficult to understand, modify and debug. The control mechanisms already mentioned provide most of the control needed. In addition a highly structured form of forward branch, the escape-expression, has been included. There are eight forms of escape; one for each control environment:

exitblock ϵ	*exitcase* ϵ
exitcompound ϵ	*exitselect* ϵ
exitloop ϵ	*exit* ϵ
exitset ϵ	*return* ϵ

Each escape expression causes control to exit from a specified control environment (a block, a loop, or a case expression, for example) and defines a value (ϵ) for it (*exit* exits from any control environment; *return* exits from a function).

Other control expressions are defined in the language but will not be discussed here.

*The symbol pairs **set - tes** and **nset - tesn** are somewhat arbitrarily chosen bracketing devices which delimit the set of choices in **case** and **select** expressions.

3. Data Structures

In order to satisfy the objectives set out earlier concerning the representation of data structures, *no* implicit structures are included in Bliss. Instead, mechanisms are provided for defining representations algorithmically (that is, specifying the access method for elements of the structure), for associating particular representations with particular identifiers, and for invoking the access algorithm associated with an identifier. The definition of a representation scheme is made by a declaration of the form

structure <scid>[<formal parameter list>] = $\in$

The <scid> in this declaration, called a structure class identifier, may then be used to associate the accessing algorithm, denoted $\in$ above, with specific identifiers by another declaration

map <scid> <idchuck>

(where an <idchuk> is a sequence of identifiers, <id>'s, separated by colons) each of which is to be associated with <scid>. Once the association between a variable identifier and a structure representation has been established, the name-form "<id>[$\in_1, \in_2, \ldots, \in_n$]" becomes valid, and denotes invocation of the access algorithm defined in the associated *structure* declaration (with an appropriate substitution of actual for formal parameters). Thus the syntactic device "<id>[$\in_1, \in_2, \ldots \in_n$]" denotes a name (a pointer) resulting from the evaluation of a user-defined expression.

Consider the following example:

```
begin
  structure array2[i,j] = (.array2+.i*10+.j);
  own x[100],y[100],z[100];
  map array2 x:y:z;
  .
  .
  .
  x[.a,.b] ← .y[.b,.a];
  .
  .
  .
end;
```

In this example a very simple structure representation, array2, for two dimensional (10+10) arrays, is introduced. The structure is to be represented by storing rows, and row elements, in contiguous memory locations; we declare three segments with names "x", "y", and "z" bound to them; and the structure class "array2" is associated with these names. The syntactic forms "x[$\in_1, \in_2$]" and "y[$\in_3, \in_4$]" are valid within this block and denote names resulting from evaluation of the accessing algorithm defined by the array2-*structure* declaration (with an appropriate substitution of actual for formal parameters).

For purposes of exposition (though it's not implemented this way) one may think of the structure declaration as defining a function which takes both the name of an instance of a structure and its accessing parameters as arguments. The structure declaration in the previous example,

structure array2[i,j] = (.array2+.i*10+.j);

is conceptually identical to a function declaration

function array2(f0,fl,f2) = (.f0+.fl*10+.f2);

The expression "x[.a,.b]" and "y[.b,.a]" correspond to calls on this function, i.e., to "array2 (x,.a,.b)" and "array2(y,.b,.a)".

Consider how the combined mechanisms of the *structure* declaration, *map* declaration, and name form "<id>[. . .]" achieve the objectives earlier set for them.

First, the programmer has complete control over the representational scheme for each of his data structures. Since names are manipulable objects in the language, any computation which is possible in the machine can be used to produce a name, and hence can be used as the access algorithm for elements of a structure. Specific properties of specific instances of a data structure, the size of its elements, the usual form of their access, etc., may be fully exploited.

Second, the specification of a representation and the algorithms which manipulate elements in the structure have been separated. The syntactic form "X[.i,3]" denotes the name of a specific element of the structure called X, independent of how that structure is represented; the representation of that structure may be changed by altering the structure declaration without changing the algorithms which operate on elements of the structure.

Efficiency

Although it has not been explicitly discussed in the preceding material, a major aspect of the Bliss effort has been to design the language in such a way that it is possible for the Bliss compiler to produce highly efficient object programs —comparable to those which a good programmer would write in assembly language. There are two facets to this aspect of the language design. First, the language had to provide natural mechanisms through which the user can gain access to the underlying hardware. In a few cases this means including specific language constructs which utilize specific hardware features; in most cases, however, it means choosing the overall structure of the language so as to mesh neatly with the underlying hardware structure. Second, the language had to be designed such that its compiler could reasonably interpret the programmer's intentions and produce "optimal" code. This aspect of the Bliss design is its most easily documentable success; examples of numeric subroutines, for example, written in Bliss generate one-half to one-third the code produced by some of the most highly touted optimizing compilers (e.g., IBM/360 Fortran H).

Experiences Using Bliss

The Bliss language has been in active use for approximately two years for a wide variety of systems, including: the Bliss compiler itself, a WATFOR-like fast Fortran compiler, an implementation of APL (a conversational programming system), a SIMULA-like discrete event simulation system, an i/o support system, an accounting system, the kernel of a small operating system, and many applications programs. Our experience using the language over the past two years, and in watching others use it on a variety of systems, gives us some contidence that we can objectively evaluate it as a tool for systems programming. For example, along the dimension of programmer productivity (measured in instructions/programmer/ day (i/p/d) of debugged code) we recently obtained the following data on some projects at CMU:

Project	Language	Size	Man-months	i/p/d
Algol	Assembly Lang.	12k	50	11
Bliss	Bliss	31k	62	23
APL	Bliss	30k	35	39
TENFOR[a]	Bliss	12k	7	78
BLIO[b]	Bliss	7k	2	159
POOMAS[c]	Bliss	7k	3	100
TECH[d]	Bliss	3k	1½	91

[a] the WATFOR-like fast Fortran compiler
[b] the i/o support system
[c] the SIMULA-like simulation system
[d] a chess-playing program

A generally accepted value for i/p/d is five for systems written in assembly language; thus, we see a productivity increase of 4 to 30 resulting from the use of Bliss. So far as we are able to determine the quality of these systems, measured by code size and speed, are comparable to (say within 10%), or surpass those written in assembly language. The quality of these systems when measured by such criteria as readability and modifyability certainly exceeds that of systems written in assembly language, but it is nearly impossible to assign quantitative values to these measures.

Looking more closely at specific features of the language which were considered "experimental" at the time of the initial design, some have been a resounding success, others have failed in one way or another. For example, the removal of the *goto,* the structure mechanism, and the "match" between the logical Bliss machine and the physical computer on which it is implemented, are counted as substantial successes. One of our notable failures was in not recognizing the need for incorporating Bliss into a "total system" including a specialized editor, debugging support, teaching aids, etc.

One outgrowth of the experience gained from the use of Bliss merits special mention—the timing package. Systems such as those written in Bliss are usually large, and quickly exceed the implementor's ability to grasp the interaction between their various parts. Even though a language such as Bliss facilitates the modification of a system to improve its performance ("tuning" it) it is seldom clear what parts of the system need such attention. Human intuition about such things is usually very poor. Therefore, a set of Bliss routines has been developed which makes dynamic measures of a system's performance and displays information such as the space and time devoted to various portions of the system, the interaction of its various components, etc. The design and further development of the timing package and other similar support tools is currently one of the major research activities centered around Bliss.

Conclusion

An attempt has been made to present a design rationale and its manifestation in Bliss. One interpretation of this rationale is an indirect definition of the systems programming problem area. In the simplest case this manifests itself in a break with the traditional interpretation of identifiers in higher-level languages, and in the consequent demand on the programmer to be consciously aware of the distinction between names and values. The structure mechanism may be interpreted as a statement of judgment as to the extreme importance of the representation, modification, and allocation issues in systems programming—and hence that these issues must be explicitly within the programmer's attention and control.

Acknowledgments

I am deeply indebted to Nico Haberman and Don Russell who participated in the initial design of the language, and to Chuck Geschke, Dave Wile, Jerry Apperson, and Ron Brender who are largely responsible for its implementation.

Reference

1. Wulf, W., *et al.*, Bliss Reference Manual, Carnegie-Mellon University, Computer Science Department, January 1970.

The Kernel Approach to Building Software Systems

Allen Newell
Peter Freeman
Donald McCracken
George Robertson

In this short essay we will discuss a possible approach to building software systems. Our interest in building systems is driven most directly by involvement in the construction of artificial intelligence systems. But building large systems of programs is a fundamental activity throughout all of computing and has independent status as a central problem in computer science. The formidable difficulties that have emerged in producing third generation software systems well illustrate the problem.

The scheme to be explored for creating software systems is based on growth from a small kernel of code and data. The approach responds to somewhat different considerations than the more widely used alternatives of macro-assemblers and higher level languages. A full treatment would require laying out the existing approaches, as currently understood, and providing a comparative analysis. The purpose of this paper, however, will be served by a characterization of what is involved in the kernel approach.

We have been experimenting on the PDP-10 for some time with a succession of kernel systems: L*(A), L*(B), ... We must emphasize that the approach is highly experimental and that substantial issues remain unresolved so that we focus here on some of the implications of using the kernel approach to system building. L*(F), the version which has received the most polishing and use,[2] will serve as an example to make our points concrete. Its specifications are summarized briefly in the Appendix.

The idea of evolving a system from a small beginning is not new. It supplies some of the fascination that computer science has always had with bootstrapping and recursion. A widespread variant, for example, is getting compilers to compile more efficient versions of themselves. The concept of the growing machine, developed by Carr and his students at Pennsylvania[1] has some of the same spirit. Also Nievergelt has built a minimal list processing system suitable as a basis for more complex list processors.[3] But probably the most explicit kernel development is an experimental system called WISP, developed a few years ago by Maurice Wilkes,[5] which stressed not only bootstrapping, but also starting with a small initial system. WISP has had some progeny[4] and possibly should be taken as the spiritual ancestor of kernel software systems. But the idea is so fundamentally attractive that undoubtedly other such systems have been created, not all of which have seen the light of publication.

The Basic Idea of a Software Kernel

A kernel software system is a small nucleus (i.e., kernel) of code and data that grows to become a larger, more complex system. The kernel provides a base for an expanding range of systems, as shown in Figure 1. The arrows in the figure represent evolution through time. The kernel evolves into a system with system building capabilities, which then evolves into a particular application system (in our case, an artificial intelligence program). We draw a tree to show how each use of the kernel system to build a new program follows a different linear line of evolution that branches off either earlier or later from its sibling systems. We have shown each system as containing the whole of the preceding system from which it grew. However, nothing prevents a final application system from being totally separate from the system that produces it (as a compiled program is distinct from its compiler).

We will finesse the question of the exact nature of software systems and of system-building systems. As a definitional matter, we can take a software system as a body of code and data that has the capabilities of producing further programs; a system-building system thus being one capable of producing further systems. By enumeration, software systems contain facilities for creating, executing, debugging, filing, editing, and managing programs. Accurate characterization of the nature of software systems is ultimately critical to the design of system-building systems, but is beyond the limits of this essay.

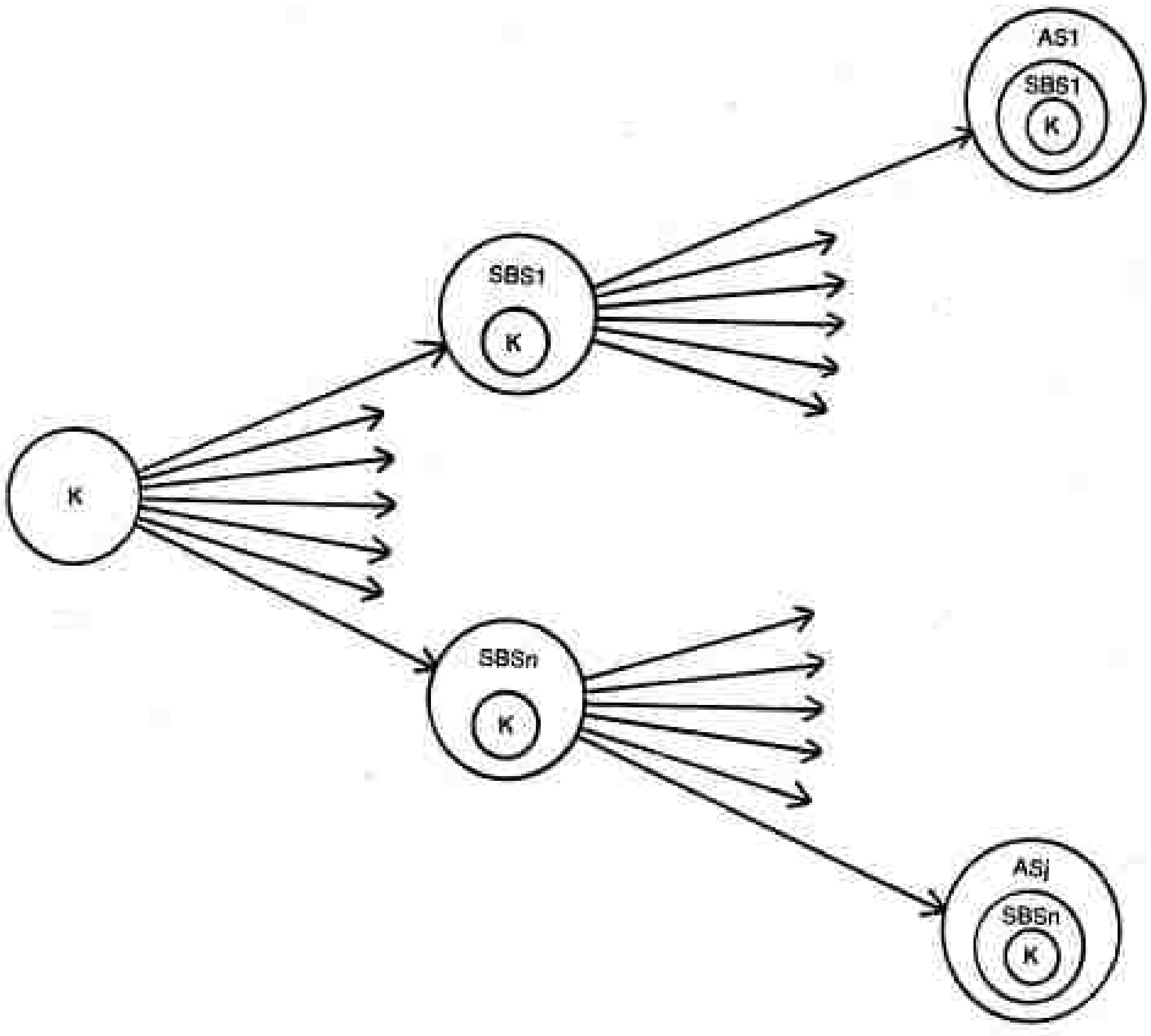
The two fundamental notions that define the design philosophy of a kernel system are small initial size and self-sufficiency for growth. Out of these two must come whatever advantages the approach has. Let us examine each in turn.

Small initial size. Large systems must eventually be grown. Thus the advantages of small size directly affect only the initial system. But small size can insure that the kernel itself can be easily constructed and completely debugged. It can insure that the user can fully understand the kernel and that radical modification is possible.

The advantages just enumerated stem not from the size of the kernel in absolute terms, but from size relative to the capabilities of human programmers. The L*(F) kernel (see Appendix) appears to be small enough to gain the advantages. For instance, the first implementation, L*(D), was constructed and debugged by one man in about five months. Subsequent versions, such as L*(F), have taken on the order of half a man-month, even though involving substantial conceptual modifications implicating more than 50% of the code. The L*(F) kernel appears to be virtually bug free after an additional man-month of polishing.

The small initial size of the kernel is essentially a commitment to simplicity as a design philosophy. Size per se does not guarantee simplicity, of course. It can be thrown away by introducing complexity and baroqueness at any juncture. Thus, a requirement of simplicity must enter into each design decision in the development of the kernel and of the strategies for growth.

Figure 1.



Kernel

Kernel System Building Systems

Application Systems

In L*(F) the kernel has the simple structure of an independent set of subroutines, each typically 5-15 instructions long. The maximum routine is 60 instructions, dictated by the need to deal with the essentially arbitrary i/o interface of the PDP-10 monitor. The depth of sub-routine nesting, which indicates the interdependency in a set of routines, is at most two in almost all cases. Uniform naming and coding conventions can be maintained throughout. Thus, the amount of information needed to comprehend any part of the kernel is small.

The advantages given above of small size and simplicity show up primarily in terms of ease of understanding and modifiability (e.g., another one is ease of producing complete and comprehensible documentation). These requirements are desirable in any system, but they are absolutely central to the kernel system approach. In many design philosophies, one strives to construct an interface with the user (consisting primarily of a higher level programming language) that is sufficiently smooth and self-contained that the user need never be concerned with the underlying structure that realizes the interface. Ease of understanding applies only to the face presented to the user. With a kernel system, each ultimate user system is potentially grown from the kernel itself (though it may initially start off from some advanced base). All aspects of the kernel must be understandable to permit the user to grow the system in ways appropriate to his own needs.

Self-sufficiency for growth. We can contrast a system that is grown by means of mechanisms internal to it with a system that is produced as a passive object by some set of external tools. A compiler is an example of an external system, as is a text editor. A kernel system is an example of a system capable of internal growth. A priori, neither design philosophy is better; it is doubtful that any such general approach can emerge as preferred for all systems building tasks. However, the two philosophies lead in different directions.

Growth means adding instructions and data, debugging them, and providing for their execution and the use of their output. Growth in this context must be extended to modification and contraction of existing structure. Indeed, the structures provided in early stages of growth are almost always deficient in some respects, e.g., efficiency, and require reshaping or replacement later.

For growth to be internal requires that all mechanisms to effect that growth must be in the kernel. This requirement can be satisfied in a highly indirect way, of course. The kernel may not contain the tools necessary for a given task, but only the tools necessary to construct these tools. Bootstrapping as a design philosophy implies the emergence of ultimate results through an extensive cycle producing tools for tools for tools.

It is doubtful whether an invariant set of initial functions must be provided by a kernel. For one thing, different machine environments will impose different functional requirements. For another, there may well be alternative bases. The set of functions provided by the L*(F) kernel, operating within the PDP-10 time-sharing monitor, serves as an example. We state these functions in general terms, though they are realized in specific ways for L*(F), as indicated in parentheses:

(1) Creation of internal symbols to designate data structures (addresses).
(2) Creation, manipulation and interpretation of a class of arbitrary symbolic expressions (lists).
(3) Input and output to a standard user interface (teletype).
(4) Creation and modification of external names to be in correspondence with internal symbols (name table).
(5) Reading and writing to secondary memory (disk files via monitor).
(6) Access to all of the entities in the kernel (preassigned external names for all kernel routines and data).
(7) Creation and recognition of arbitrary bit patterns (operations to go from lists to bits and bits to lists).
(8) Saving an existing instance of the system and restarting a copy of it at a later date (operators to evoke save and restart mechanisms in monitor).
(9) Recovery from error and exploration of the errorful instance of the system (context swapping operation to establish a viable operating context).
(10) Locus of control at the user interface top-most level of control reads input from teletype to be interpreted as program).
(11) Enough resources to avoid sudden death and access to additional resources (initial available space and operations to get more from PDP-10 monitor).

This list of functions differs considerably from that associated with a programming language, such as Algol, for the emphasis is on being a viable self-contained system. Thus, the ability to save and restart the system and to survive catastrophic error, functions normally associated with operating systems, show up as critical initial functions. The specification of locus of control at the user interface (item 10) is to be especially emphasized. This is essentially the direct mode of execution existing in many conversational programming languages (e.g., see the description of LCC in the 1969 Annual Review). With this control the user plays the role of the executive routine, executing operations singly and at will, shaping the system as required.

Many things are missing from this list that must ultimately be provided in any system-building system and some of the functions in the list are provided only in rudimentary form. For instance, access to the secondary memory is essential right at the beginning, but a rather elementary capability suffices. (In L*(F) it is read and write from a single fixed file.) Some of the missing facilities that come immediately to mind are:

(1) Editing
(2) Tracing programs
(3) General communication to monitor
(4) General storage and retrieval of files
(5) Assembly of machine routines
(6) Modification of user interface
(7) Higher level language at user interface
(8) Creation of new data types with their proper operations.
(9) Space management
(10) Error detection with diagnostics

All of these must be grown with the facilities given initially in the kernel. This involves a bootstrapping procedure in which elementary tools are built and from these still other tools are developed. For instance in the bootstrapping sequence developed for L*(F), the first thing that is done is to create ways of modifying the interpretation given to the input stream, so that some new notation can be introduced. Later in the sequence the initially existing limitation on external names to five characters (which permitted a simple name table) is lifted by replacing the entire external naming system. This new system is such that it permits the introduction, one by one, of the various notations of a higher level language at the user interface. These later notations supersede, of course, the original mechanism for achieving various notations.

It might be thought that there is a single bootstrapping sequence, but that is not the case. For example, early on an editor and an interpreter for stepping through a program under manual control are introduced. Both are very much a matter of individual design, and alternative growths of the system could dictate substantially variant schemes. Thus, the branching process indicated in Figure 1 occurs at many places along the line of development.

Consequences of the Main Ideas

The two notions—small initial size (with the more general commitment to simplicity) and self-sufficiency for internal growth—constitute the central design philosophy of a kernel system. But consequences for additional features of design philosophy flow from them. Some of these need not be followed, for in growing a kernel into a larger system it can be shaped ultimately to quite divergent strategies. But these additional characteristics are consonant with the basic philosophy and serve to exploit it.

Total accessibility. The point of growing a large system from a small beginning is to permit all aspects of the system to be shaped by the ultimate needs. Any aspect of a pre-existing system can become a limiting factor on the efficiency or abilities of a final user system. It has always been an important consideration in system-building systems that they have complete access to the total facilities of the target machine. This at least provides the potentiality of designing user systems with maximum efficiency and capability. The continued popularity of assemblers is due in part to the transparent way in which they provide complete access to the machine's facilities.

Total accessibility is a combination of requirements, some on ease of understanding of the system, some on the available means for constructing new systems. The emphasis earlier on small initial size and simplicity indicates how a kernel system deals with requirements on ease of understanding. We are concerned here with the means for constructing new systems. One aspect is access to the kernel itself. In L*(F) this is provided by having external names for all the routines and data structures in the kernel and by having all of the kernel in the address space (i.e., internal symbols to designate every cell in the kernel).

The most important aspect of total accessibility is being able to make use of all of the machine's basic facilities. One course is to mirror each feature of the underlying machine in the operations of the kernel (in the manner of an assembler). Give the complexity of current machines (e.g., many instruction types and hundreds of individual instructions) this conflicts strongly with the requirements of simplicity and small initial size. Attempting to accomplish it in the initial system anyway forces an impoverished and minimal scheme. Assemblers show just such impoverishment in comparison with higher level language systems. The solution used in L*(F) is to delay access to the total machine until later in the evolution. The provision in the kernel of a basic bit facility (item 7) yields the logical capability to lay down arbitrary code and the provision of the general symbolic manipulation system (item 2) yields the potential for constructing linguistically suitable schemes for designating new instructions, routines and data types.

Integrated programming environment. Conversational language systems, such as Joss and LCC, have moved toward providing all computing functions—program definition, editing, execution, debugging, storage and retrieval on files, etc.—with a common language. It is no longer necessary in such systems continually to change between distinct (and generally non-cooperating) subsystems, each with their own conventions, to perform the different tasks associated with building programs. This is called an integrated programming environment and its desirability is widely recognized.

Integrated environments occur only rarely outside of specialized conversational systems. The design philosophy inherent in most operating systems encourages a plurality of distinct processors and languages, e.g., assemblers, editors, higher languages, the command language, file systems, etc. Most system building is accomplished in this kind of an environment.

The kernel system leads naturally to an emphasis on attaining an integrated programming environment. It is consonant with the already established goal of simplicity. Since the eventual system has roots all the way back to the kernel, there is less inclination for a layered structure of system and subsystem to grow up, which is the genesis of multiple language systems. If a partially grown system appears to be congealing into clusters of subsystems with distinct conventions, then the system can be regrown from an earlier point and shaped to a more homogeneous form.

Modest experience with L*(F) indicates that a unified environment may be rather easily achieved, mainly because at appropriate stages of growth the adding of new facilities with diverse functions requires only small amounts of new program. The initial language of L*(F) is a simple list language called L*L, which is written horizontally and whose coding density is not unlike that of LISP. In this language a simple online editor requires a few lines; a monitor for manually stepping through programs requires less than a page of code; a system for entering text requires less than half a page. All these come along rather early in the bootstrapping sequence, though not before some other tools have been constructed. A rudimentary compiler to transform L*L expressions into machine calls (thus eliminating the interpreter) requires less than two pages of code and a primitive assembler without macro features (unneeded in L* with its general symbolic capabilities) requires about two pages. Some of the simplicity of these systems arises from the design philosophy of creating an integrated programming environment. Each augmentation relies on the existing mechanisms and conventions, being just the necessary growth to provide the additional functions.

Multiple use of structure. Using one structure to provide the same function throughout a system is a common practice. When systems are built up from a kernel the possibility of multiple use exists along another direction. The structures that provide a function in the system-building system can also provide that function in the ultimate application system. For example, the syntax analyzer used in the system-building system can also become the syntax analyzer in the final user system. (It is a rare compiler that can transfer its own syntax analyzer to an object program for use there.)

Several effects follow from this multiple use of structure. It contributes to the maintenance of simplicity and it undoubtedly provides some savings in space. But the main effect is the use of already understood and debugged components in the creation of a new system.

The ability to use structures in this way arises from the commitment of internal growth, which thoroughly blurs the line between system-building system and application system. New systems are built, not by using a finished system-building system (grown, say, by someone else), but by backing up and starting off again in a new direction from an earlier point.

L*(F) has an additional design feature that serves to enhance the extension of existing structures to new contexts. The data structures (and the internal symbols which designate them) form a discrete set of types. Differential action depending on the type of data being processed is both fast and convenient. For example, a natural organization for a print routine is as a set of intercommunicating print routines, one for each type of data.

The kernel contains only a small set of initial types, but new types can be added at will. In fact, one of the main techniques for growth is the addition of types. Extension encourages fitting new subsystems into the existing framework of types, thereby extending existing facilities. An illustration would be the creation of a type LISP, fitting into the existing interpretive structure (which is by type) and thus permitting all the existing editing, debugging, and printing facilities to be immediately used in a LISP context.

Personalization. The notion of a system-building system encourages producing a multitude of systems adapted to special circumstances. The notion of the kernel system enhances this to the point of considering personalized systems to be the rule. Personalization is not simply the institutionalization of the natural tendencies of systems programmers to be idiosyncratic. Rather, it means that a system should be built as much as possible to serve the individualized needs of potential users.

Rigid standardization of programming systems up to some high level of development (as in Fortran) has some important advantages. Common systems over large populations of users permit communication of programs and ideas and lessens the need of continually learning new systems. However, for the types of systems built in the course of computer science research, these advantages are often outweighed by the need to shape the system to new demands and concepts. Similarly, extracting the utmost efficiency from a machine, which implies the complete adaptation of the system to the task at hand, also can outweigh the advantages of communality.

The kernel itself, having been carefully designed once and for all, might seem to be exempted from personalization. But this is not the case. It can, of course, be modified as we have emphasized in discussing accessibility. Since it is relatively small, however, the final system can simply be grown away from any of its conventions. For instance, a new language, such as LISP, could be introduced to completely supersede L*L, the initial language provided in the kernel.

Design iteration. Iteration of a design is generally considered a laudable goal. In large software systems (as in some other areas) it rarely happens, primarily because of the extensive effort involved and the inevitable occurrence of unanticipated difficulties that stretch out the construction effort beyond all preset deadlines. Typically, design iteration occurs at the specification stage, then the design is committed and an initial version is brought into being. Once in use, modifications and revisions are made (in successive "releases" of the system) attempting to adapt the system to the actual environment.

The kernel system approach appears to lead to a highly iterative design style. Systems are grown and regrown from early points in the tree of development (Figure 1), and design becomes an experimental activity rather than an analytic one. Part of the reason for this design style is certainly the small initial size, which of course must eventually give way to mature systems of large size. But part of the reason also appears to be that the growing system contains substantial investment in structures that help to grow the system further. Thus easy regeneration of the system from early stages becomes an important subgoal in the design of a system.

Our experience in L*(F) on iterative design of large application systems through regeneration is still minimal. We do have experience about iteration of design for the kernel itself. So far during the year in which we have devoted substantial effort to L* we have brought six systems into full existence: L*(D) through L*(G) on the PDP-10 and L*11(A) and L*11(B) on the PDP-11. Each of these has explored basic variations in the design space of kernel systems. Several more iterations seem indicated at present before we will finally know enough about the kernel to create a final system, all of whose further modifications should arise through internal growth.

Summary. All of these aspects of system design just discussed are consonant with the idea of a kernel system building system. However, they are not essential. It is quite possible that highly successful lines of development would shun some of them entirely. One could start with a kernel, construct a particular application system with its own programming language, file system, etc., discarding all the system-building scaffolding so that no trace of it remained in the final application system. But these notions of personalization, design iteration, multiple use of structures, etc., appear to be the ideas to be exploited to make the kernel approach to system building viable.

The Essential Problems

So far we have defined the essential nature of the kernel system approach to system building and explored the various directions in which its advantages might lie. But there are some difficulties, too. A kernel system depends on getting most of the ultimate facilities for system building indirectly, providing only the tools for their construction (and sometimes being even more remote). Looking at current schemes for building systems, four main things have been provided by one or another system: (1) direct and transparent access to the underlying machine; (2) higher level languages; (3) efficient code production and (4) supporting facilities, predefined and working, evocable through a command language. None of these is available directly in the kernel; all have to be built.

The essential tension (to use a favorite phrase of Bill Wulf's) is between the effort to be spent in building up these facilities and the advantages of the user having shaped them himself to his own needs. On the effort side must be counted not only the programming and debugging required, but also the intellectual investment by the user in understanding the functions to be built and the mechanism that will realize them. It may turn out that users simply will not wish to understand all the subsystems involved in their application system. More likely, the use of kernel systems will be restricted to professional systems programmers and will not be a tool for the casual user. Our own intended use, to build artificial intelligence systems, certainly is of this sort.

Success of the kernel system approach demands that all of the ultimate facilities of a system-building system (editors, translators, higher level languages, etc.) be obtainable without undue pain and effort. Otherwise it is surely not worth anyone's while to work through their construction. Given that kernels start with so little and given the general experience with how much work it is to build systems programs, this appears highly implausible on the face of it. The distance between the initial point and a fully developed system simply appears to be too great. We did note some positive evidence earlier from L*(F)—that subsystems require very little code. But the evidence is far from conclusive yet. That the distance really can be covered with ease is undoubtedly the most important hypothesis underlying the use of kernels for system-building.

Conclusion

We have presented briefly the idea of growing software systems from a small kernel of code. The basic idea incorporates both small initial size and self-sufficiency for internal growth. We fleshed out these two ideas by describing several associated elements of design philosophy. Throughout we have tried to present the basic notions independent of our particular line of experimental systems (L*), though using it as a source of illustrations. We have sought in this way to draw attention to another alternative for system-building that seems to have some promise.

References

1. Carr, John W. III, *Growing Machine Handbook*, Moore School of Electrical Engineering, University of Pennsylvania, 1967.
2. Newell, A. D. McCracken, G. Robertson, and L. DeBenedetti, "L*(F)", Computer Science Department, Carnegie-Mellon University, January, 1971.
3. Nievergelt, J. F. Fischer, M. I. Irland, and J. R. Sidlo, "Nucleol—A Minimal List Processor," Department of Computer Science, University of Illinois, Report No. 324, April, 1969.
4. Waite, W. M., "The Mobile Programming System: STAGE2," *Comm. ACM*, vol. 13, no. 7, p. 415-421, July, 1970.
5. Wilkes, M. V., "An Experiment with Self-compiling Compiler for a Simple List-processing Language," *Annual Review in Automatic Programming*, vol. 4, Richard Goodman (ad.), Macmillan, New York, 1964.

Appendix: Description of L*(F) Kernel

L*(F) is a kernel system operational on the PDP-10.[2] Table 1 lists some of its main characteristics. Space prohibits discussing these in detail, but we can comment on a few of them that relate to the central design idea.

First is the adoption of a simple list language (called L*L) as the initial language. As is well known, only a few primitive actions suffice for an essentially complete list facility for doing symbolic manipulation.

A second important feature is the use of a homogeneous symbol system. Symbols are taken to be addresses and are used to refer to everything. For instance, there is a symbol (an address) corresponding to each of the 128 characters.

A third major mechanism is a universal type system. Every symbol has a type, which describes the nature of the data structure it designates. Initially, only a minimum number of types are provided: list (the basic data structure), program list (list to be interpreted as programs), Integer (required for incrementing and differencing addresses), machine (to identify machine code), character (the fixed set of 128 symbols), and cells (everything else). Everything has a type, even the registers of the machine (type list and type cell), the size of the symbol table (type integer) and each instruction in the kernel (type machine). Types are totally dynamic: the type of any symbol can be changed, new types can be created, the functions associated with a type can be changed, the total number of types in the system can be increased (or decreased).

Action can be type dependent everywhere, so there is always available a relevant discrimination that can be used to direct processing. Thus, print routines operate conditionally on the type of the symbol to be printed. To obtain the advantages of a type system one needs to have type dependent action as fast as possible (for it is like an inner loop calculation) and for the structure that holds types to impose no constraint on the types of data structures. In L*(F) we pay a very high price for this: for each cell of the system an extra cell is taken to hold the type index. Although the space cost is substantial (being reminiscent of the space-cost paid for list structures), the gains appear to be impressive.

One feature of L*L is worth mentioning: the principle of semantic interpretation. That is, to interpret a symbol in a program list the interpreter associated with the type of that symbol is executed. This means that a program list itself has no syntactic structure. For example, if LIST and SYMB are respectively a list and symbol of type list and TEST is a program, then the program list

(LIST SYMB TEST)

results in TEST being interpreted as a program with LIST and SYMB as operands. This happens because each of the three symbols is interpreted in order, but the interpreter associated with LIST and SYMB, being interpreter for type list, treats these symbols as operands, whereas the interpreter for TEST treats it as a program. If TEST were type machine, then it would be executed as a machine routine.

The kernel of L*(F) does not contain direct access to all the features of the PDP-10 machine. Thus, ultimate access is obtained by providing tools in the kernel. These are two special interpreters, one of which deposits a set of bits in a field in a word, the other of which extracts a set of bits from a word. These interpreters can be associated with various types of symbols. Initially, type character is the only type for which these instructions make sense, permitting the packing and unpacking of character strings simply by interpreting lists of characters.

TABLE 1: CHARACTERISTICS OF L*(F)

Size of kernel:
- 1078 instructions, 924 data words, 7139 with all available space.

Symbol system:
- Homogeneous system of symbols;
- Symbols are addresses;
- All addresses in main segment are symbols.

Type system:
- All symbols have types;
- Types form a small discrete unstructured set (initially 16);
- Type dependent action is immediate;
- The type of a symbol is completely dynamic;
- The function of a particular type is dynamic (slowly);
- Mechanized by putting type index for each symbol (address) in a corresponding word in a special segment (trades space for time).

Initial types:
- Lists, program lists, machine, integers, characters, cells.

Initial language:
- L*L (a simple list language of type program list);
- interpretive (base rate = 30,000 cycles per second);
- Interpretation by executing interpreters associated with each type:
  - For program lists: interpret each symbol of list in turn;
  - For machine: Execute machine routine;
  - For others (data): Push symbol into data stack;
- Operand and result communication via data stack;
- Operator/operand distinction made by type of symbol (i.e., by interpreter used for type);
- Control structure:
  - Loops handled by repeated interpretation of program lists;
  - Action can be conditional on top of data stack;
    - No direct transfers (no "goto's");
  - Effective style of language is Polish post-fix.

Symbol table:
- Entries in initial table limited to arbitrary 5 character names;
  - Uses sequential search for lookup;
- Number of names expandable dynamically;
- Size of names not easily expandable;
- Table scheme is replaceable.

Accessibility:
- All relevant names in kernel entered into L* symbol table;
- All routines in kernel executable from within L*L;
- All internal structure of L*L of type list (data and routine stacks);
- Kernel has simple structure:
  - 85 almost independent routines (maximum nesting, 2 in almost all cases);
  - Standard scheme for communication between machine routines and environment (including L*L stacks);
- Format lists to deposit and extract bits within words.

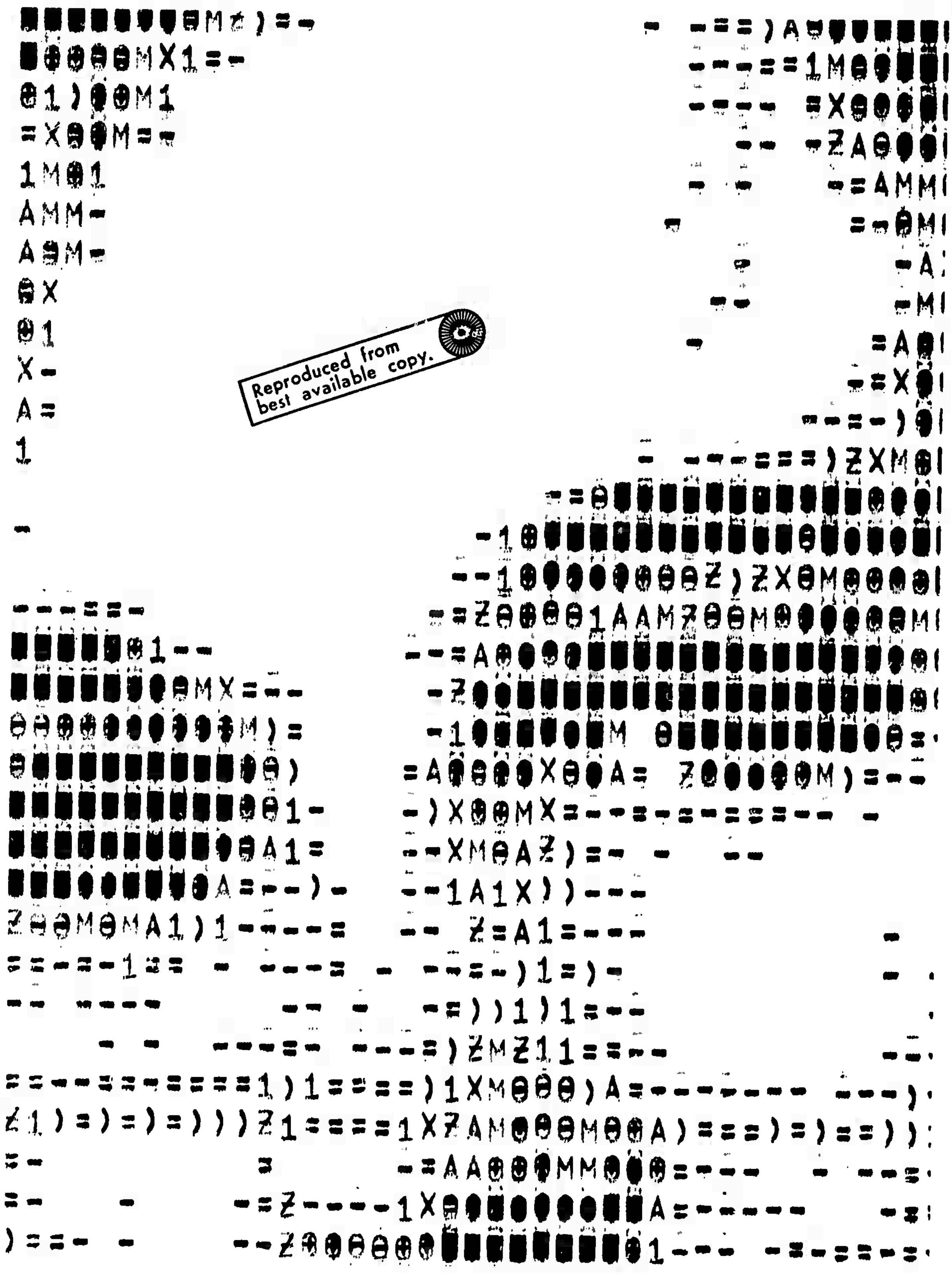
Interfaces to environment:
- Interfaces exist to teletype and disk (via Monitor conventions);
- Communication via list of characters (i.e., symbols of type character);
- Processes by using special interpreter for type character (executes L*L process associated with each character).

Style of use:
- Conversational (via PDP-10 Monitor).

Debugging and recovery:
- Swap mechanism to reinstate functioning context and make buggy context available for investigation and correction dynamically.

Space management:
- Automatic for lists and single cells;
- No recovery when exhausted (but creatable);
- Access to PDP-10 Monitor for more space.

Coding of kernel:
- In Macro-10, the PDP-10 macroassembler.



Faculty

C. Gordon Bell
Professor of Computer Science and Electrical Engineering
S.B., Massachusetts Institute of Technology (1956)
S.M., Massachusetts Institute of Technology (1957)
Carnegie, 1966: Computers and Computer Networks

Robert N. Chanon
Instructor
B.S., Carnegie-Mellon University (1967)
Carnegie, 1968: Programming

Lee Erman
Research Associate
B.S., University of Michigan (1966)
M.S., Stanford (1968)
Carnegie, 1970: Artificial Intelligence

Arie N. Habermann
Associate Professor of Computer Science
B.S., Free University, Amsterdam (1953)
M.S., Free University, Amsterdam (1957)
Ph.D., Technological University, Eindhoven, The Netherlands (1967)
Carnegie, 1968: Operating Systems and Programming Languages

Per Brinch Hansen
Visiting Research Associate
B.S., Technical University of Denmark
M.S., Technical University of Denmark
Carnegie, 1970: Software

Donald W. Loveland
Associate Professor of Computer Science and Mathematics
B.A., Oberlin College (1956)
S.M., Massachusetts Institute of Technology (1958)
Ph.D., New York University (1964)
Carnegie, 1967: Logic (Recursive Function Theory), Mechanical Theorem Proving, Computational Complexity

Philip H. Mason
Instructor
B.S., Carnegie-Mellon University (1967)
Carnegie, 1969: Systems Programming

John W. McCredie
Assistant Professor of Computer Science
B.E., Yale University (1962)
M.S.E.E., Yale University (1964)
M.S.I.A., Carnegie-Mellon University (1966)
Carnegie, 1968: Simulation, Optimization Techniques and Systems Analysis

Ugo Montanari
Visiting Research Associate
Carnegie, 1970: Artificial Intelligence

Richard Neely
Research Associate
B.A., University of Oregon (1966)
M.S., Stanford University (1968)
Carnegie, 1970: Artificial Intelligence

Allen Newell
University Professor
B.S., Stanford University (1949)
Ph.D., Carnegie Institute of Technology (1957)
Carnegie, 1961: Artificial Intelligence, Simulation of Human Thinking, Programming Languages

David L. Parnas
Associate Professor of Computer Science
B.S., Carnegie Institute of Technology (1961)
M.S., Carnegie Institute of Technology (1964)
Ph.D., Carnegie Institute of Technology (1965)
Carnegie, 1966: Simulation, Automatic Design of Finite Automata, Computer Languages, Computer System Design

Alan J. Perlis
Professor of Mathematics and Head of the Department of Computer Science
B.S., Carnegie Institute of Technology (1943)
M.S., Massachusetts Institute of Technology (1949)
Ph.D., Massachusetts Institute of Technology (1950)
Ph.D. (Hon.), Davis and Elkins College (1968)
Carnegie, 1956: Programming Languages

D. Raj Reddy
Associate Professor of Computer Science
B.E., University of Madras (1958)
M. Tech., University of New South Wales (1961)
M.S., Stanford University (1964)
Ph.D., Stanford University (1966)
Carnegie, 1969: Artificial Intelligence and Man-Machine Communication

Ronald M. Rutledge
Assistant Professor of Computer Science
S.B., University of Georgia (1957)
S.M., University of Georgia (1960)
Ph.D., University of Tennessee (1964)
Carnegie, 1968: Computer Science Management, Measurement and Evaluation of Operating Systems

Herbert A. Simon
Richard King Mellon Professor of Computer Science and Psychology, Associate Dean of the Graduate School of Industrial Administration
A.B., University of Chicago (1936)
Ph.D., University of Chicago (1943)
D.Sc. (Hon.), Case Institute of Technology (1963)
D.Sc. (Hon.), Yale University (1963)
LL.D. (Hon.), University of Chicago (1964)
Fil.D. (Hon.), University of Lund, Sweden (1968)
Carnegie, 1949: Computer Simulation of Cognitive Processes, Artificial Intelligence, Management Science

William A. Wulf
Assistant Professor of Computer Science
B.S., University of Illinois (1961)
M.S.E.E., University of Illinois (1963)
D.Sc., University of Virginia (1968)
Carnegie, 1968: Systems Programming

Yechezkel Zalcstein
Assistant Professor of Computer Science
A.B., University of California, Berkeley (1962)
M.A., University of California, Berkeley (1965)
Ph.D., University of California, Berkeley (1968)
Carnegie, 1969: Automata Theory, Algebraic Theory of Linear Systems, Finite Semigroups

Departmental Staff

Engineering

William Broadley—Manager of Engineering Design and Senior Research Associate
Paolo Coraluppi—Engineer
Ralph De Lucia—Engineer
Roland Findlay—Technician
Christopher Hausler—Technician
Paul Newbury—Engineer
Kenneth Stupak—Technician
Jackson Wright—Design Engineer

Office Staff

Roberta Gray—Business Administrator
Dorothy Josephson—Technical Typist
Georgette Katona—Secretary to Dr. Perlis
Mercedes Kostkas—Secretary
Mildred Sisko—Secretary to Dr. Newell

Operations

Carolyn Lisle—Manager of Computer Operations
Barbara Anderson—Lead Operator

Programming

Harold Van Zoeren—Manager of Programming
Howard Wactlar—Supervisor of Special Programming Projects
Diana Bajzek—Programmer
Donald McCracken—Programmer
George Robertson—Programmer
David Wile—Junior Research Scientist

Graduate Students

Agarwal, Durga
B.E., Birla Institute of Technology and Science (1969)
Electronics
M.Tech., Indian Institute of Technology (1970)
Computer Science

Apperson, Jerry
B.A., University of Virginia (1965)
Mathematics

Ariely, Gideon
B.A., Hebrew University (1969)
Mathematics-Computer Science

Aygun, Birol
B.S.M.E., Newark College of Engineering (1965)
M.S., Columbia University (1968)
Mathematical Methods in Engineering and Operations Research

Barbacci, Mario
B.S., U.N.I. (Lima, Peru) (1966)
Electrical Engineering
Engineer, U.N.I. (Lima, Peru) (1968)
Electrical Engineering

Bauer, Madeline
A.B., Cornell University (1968)
Mathematics
M.A., University of Michigan (1970)
Computing and Communications Sciences

Berliner, Hans
B.A., George Washington University (1954)
Psychology

Bhatia, Sushil
B.Tech., Indian Institute of Technology (1966)
Electrical Engineering
M.S., Carnegie-Mellon University (1969)
Electrical Engineering

Chang, Hsiau-Chung
B.S., National Taiwan University (1971)
Physics

Chen, Robert
B.S., Rensselaer Polytechnic Institute (1966)
Electrical Engineering
S.M., Massachusetts Institute of Technology (1968)
Electrical Engineering

Cohen, Ellis
B.S., Drexel Institute of Technology (1970)
Mathematics

DeBenedetti, Lydia
B.A., Oberlin College (1967)
Economics

Dills, John
B.S., Clarkson College (1968)
Mathematics

Evans, Steven
B.A., Tulane University (1965)
Mediaeval German and Mathematics

Farley, Arthur
B.S., Rensselaer Polytechnic Institute (1968)
Mathematics

Fennell, Richard
B.S., Rensselaer Polytechnic Institute (1969)
Physics

Gerhart, Susan
B.A., Ohio Wesleyan University (1965)
Mathematics
M.S., University of Michigan (1967)
Communication Sciences

Geschke, Charles
A.B., Xavier University (Cincinnati, Ohio) (1962)
Latin
M.S., Xavier University (Cincinnati, Ohio) (1963)
Mathematics

Gillogly, James
B.A., UCLA (1967)
Mathematics

Goldberg, Henry
S.B., Massachusetts Institute of Technology (1968)
Mathematics

Grove, Richard
B.S., Carnegie Institute of Technology (1964)
Mathematics
M.S., Carnegie Institute of Technology (1965)
Mathematics

Huen, Wing Hing
B.S., University of Hong Kong (1966)
Physics
M.S., University of Alberta (1969)
Computer Science

Johnsson, Richard
B.E., Vanderbilt University (1970)
Electrical Engineering

Jones, Anita
B.A., Rice University (1964)
Mathematics
M.A., University of Texas (1966)
English

Kedar, Eliahu
B.Sc., Hebrew University of Jerusalem (1968)
Mathematics and Physics

Kendziora, Alois
B.A., Gannon College (1962)
Mathematics
M.A., University of Detroit (1964)
Mathematics

Knudsen, Michael
B.S., Pennsylvania State University (1966)
Engineering Science
S.M., Massachusetts Institute of
Technology (1968)
Electrical Engineering

Krutar, Rudolph
B.S., Carnegie Institute of Technology (1966)
Mathematics

Lee, Sai-Ming
B.A., UC at Berkeley (1970)
Mathematics-Computer Science

Lee, Tih-Ming
B.S., Tamkang College (Taiwan) (1966)
Mathematics

Lieberman, Robert
B.S., SUNY at Stony Brook (1968)
Mathematics

Linstrom, Gary
B.S., Carnegie Institute of Technology (1965)
Mathematics
M.S., Carnegie Institute of Technology (1965)
Mathematics

Lipton, Richard
B.S., Case Western Reserve (1968)
Mathematics

Lowerre, Bruce
B.S., Case Institute of Technology (1965)
Chemistry
B.S., Case Western Reserve (1970)
Mathematics

Lunde, Amund
M.Sc., University of Oslo (1966)
Mathematics

Mann, William
B.S., Lehigh University (1956)
Electrical Engineering
M.E.A., George Washington University (1964)
Engineering Administration

McConnochie, John
B.A., Dartmouth College (1964)
Mathematics

Mitchell, James
B.Sc. (Hon.), University of Waterloo (1966)
Mathematics

Moore, James
S.B., Massachusetts Institute of
Technology (1964)
Mathematics

Moran, Thomas
B.Arch., University of Detroit (1965)
Architecture

Moyles, Dennis
B.S., Carnegie-Mellon University (1970)
Mathematics

Newcomer, Joseph
B.A., St. Vincent College (1967)
Mathematics

Ohlander, Ronald
B.S., St. Mary's College (1962)
Psychology

Pfefferkorn, Charles
B.S., Carnegie Institute of Technology (1964)
Physics

Pierson, Charles
B.S., Carnegie-Mellon University (1970)
Mathematics

Pollack, Frederick
B.S., University of Florida (1970)
Mathematics

Price, William
B.A., Lehigh University (1969)
Mathematics

Rege, Satish
B.Tech., Stevens Institute of Technology (1968)
Electrical Engineering
M.S., University of Pittsburgh (1969)
Electrical Engineering

Rinde, Joseph
B.S., Stevens Institute of Technology (1968)
Mathematics

Rizzo, Michael
B.S., Rensselaer Polytechnic Institute (1968)
Mathematics

Schlesinger, Steven
B.A., Cornell University (1968)
Mathematics

Schneider, Edward
B.S., Carnegie-Mellon University (1970)
Mathematics

Shaw, Mary
B.A., Rice University (1965)
Mathematics

Shu, Hou-Shing
B.S., Taiwan University (1969)
Physics

Snyder, Larry
B.A., University of Iowa (1968)
Mathematics

Stickel, Mark
B.S., University of Washington (1969)
Mathematics
M.S., University of Washington (1971)
Computer Science

Sung, David
B.S., National Taiwan University (1968)
Mathematics

Teitelbaum, Ray
S.B., Massachusetts Institute of
Technology (1964)
Mathematics

Vasudevan, Narayanan
B.S., Engineering College (Madras) (1966)
Electrical Engineering
M.Tech., Indian Institute of Technology (1969)
Electrical Engineering

Weinstock, Charles
B.S., Carnegie-Mellon University (1970)
Mathematics

Yuo, Peter
B.S., National Taiwan University (1968)
Mathematics

Publications

Bell, C. G., "Minicomputer Architecture and Design," *Proc. Institute of Electrical and Electronics Engineers*, March, 1971.

Bell, C. G., "Some Network-Space Dimensions," *Ekistics, 30,* No. 1979, 270-271.

Bell, C. G., R. Cady, H. McFarland, B. Delagi, J. O'Laughlin, R. Noonan, and W. Wulf, "A New Architecture for Mini-Computers—the DEC PDP-11," *AFIPS—Conference Proceedings*, Spring Joint Computer Conference, 1970, 657-675.

Bell, C. G., D. Casasent, and R. Hamel, "The Use of the Cache Memory in the PDP-8/F Minicomputer," Spring Joint Computer Conference, 1971.

Bell, C. G., and J. Grason, "The Register Transfer Module Design Concept," *Computer Design, 10,* No. 5, 87-94.

Bell, C. G., J. Grason, S. Mega, R. Van Naarden, and P. Williams, "Register Transfer Modules (RTM) for Higher Level Digital Systems Design," *Proc. Purdue 1971 Symposium on Applications ol Computers to Electrical Engineering Education,* 163-166.

Bell, C. G., A. N. Habermann, J. McCredie, R. Rutledge, and W. Wulf, "Computer Networks," *Computer*, Sept./Oct. 1970, 13-23:

Bell, C. G., and J. McCredie, "The Impact of Minicomputers on Simulation—An Overview," *Simulation,* March, 1971.

Bell, C. G., and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

Bell, C. G., and A. Newell, "The PMS and ISP Descriptive Systems for Computer Structures," *AFIPS—Conlerence Proceedings*, Spring Joint Computer Conference, 1970, 351-374.

Berliner, H., "Experiences Gained in Constructing and Testing a Chess Program," *Proc. IEEE Symposium on System Science and Cybernetics*, Pittsburgh, Oct., 1970.

Berliner, H., "United States Computer Chess Championship," *SIGART Newsletter*, No. 19, Dec., 1970.

Erman, L., and D. R. Reddy, "Implications of Telephone Input for Automatic Speech Recognition," *Proc. Seventh International Congress on Acoustics*, Budapest, 1971.

For other references by L. Erman, see D. R. Reddy *et al.*

Habermann, A. N., "An Operating System Modeled as a Set of Interactive Processes," Fifth Annual Princeton Conference on Information Sciences and Systems, March, 1971.

Habermann, A. N., "An Undergraduate Course on Operating System Principles," Cosine Task Force of the Cosine Committee of the Commission on Education of the National Academy of Engineering, 2101 Constitution Ave., Washington, D.C., Spring, 1971.

For other references by A. N. Habermann, see C. G. Bell *et al.*, and W. A. Wulf *et al.*

Loveland, D., "A Linear Format for Resolution," *Lecture Notes on Mathematics* (Symposium on Automatic Demonstration) *125,* Springer-Verlag, 1970, 147-162.

Loveland, D., "A Unifying View of Some Linear Herbrand Procedures," submitted to the *Journal ol the Association lor Computing Machinery* (based on CMU report "Some Linear Herbrand Proof Procedured: an Analysis," December, 1970).

McCredie, J., "The Structure of Discrete Event Simulation Languages," *Summer Simulation Conference Proceedings*, June, 1970, 88-98.

McCredie, J. and S. Schlesinger, "A Modular Simulation of TSS/360 *Fourth Conference on Applications of Simulation Proceedings*, Dec., 1970, 220-206.

For other references by J. McCredie, see C. G. Bell *et al.*

Montanari, U., and D. R. Reddy, "Computer Processing of Natural Scenes: Some Unsolved Problems," *Proc. AGARD Symposium on Artificial Intelligence*, Rome, 1971.

Moran, T., "A Model of Multi-Lingual Designer," *Emerging Methods in Environmental Design and Planning*, Gary T. Moore, ed., MIT Press, 1970, 69-78.

Moran, T., "(ARTIFICIAL, INTELLIGENT) ARCHITECTURE: Computers in Design," *Architectural Record*, March, 1971, 129-134.

Neely, R., and D. R. Reddy, "Speech Recognition in the Presence of Noise," *Proc. Seventh International Cingress on Acoustics*, Budapest, 1971.

For other references by R. Neely, see D. R. Reddy *et al.*

Newell, A., "Remarks on the Relationship Between Artificial Intelligence and Cognitive Psychology," in R. Banerji and J. D. Merarovic (eds.), *Theoretical Approaches to Non-Numerical Problem Solving*, Part IV, Springer-Verlag, New York, 1970, 363-400.

Newell, A., O. Barnett, J. R. Cox, M. V. Mathews, and B. Waxman, "Biology and the Future Man," in P. Handler (ed.), *Digital Computers in the Lite Sciences*, Chap. 14, Oxford University Press, 1970.

For other references by A. Newell, see C. G. Bell *et al.*, and H. A. Simon *et al.*

Parnas, D. L., "More on Simulation Languages and Design Methodology for Computer Systems," *Proc. SJCC*, 1969, 739-743.

Parnas, D. L., "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," *Proc. 1969 National ACM Conference*, 379-386.

Parnas, D. L., "On Simulating Networks of Parallel Processes in Which Simultaneous Events May Occur," *Communications of the ACM*, 1969, 519-531.

Parnas, D. L., "The Application of Modelling to System Development and Design," Papers of International Computing Symposium, ACM European Chapters, 1970, 137-147.

Parnas, D. L., Review of "Dynamic Protection Structures" [B. W. Lampson, *AFIPS—Conference Proceedings*, Fall Joint Computer Conference, 1969], *Computing Reviews*, Jan., 1971.

Parnas, D. L., Review of "Productivity of Multiprogramming Computers—Progress in Developing an Analytic Prediction Method," [D. J. Lesser, *Communications of the Association for Computing Machinery 12*, 12], *Computing Reviews*, Jan., 1971.

Parnas, D. L., P. J. Courtois, and F. Heymans, "Concurrent Control with Readers and Writers," Report R143 of the MBLE Laboratoire de Recherches, Brussels, Belgium, September, 1970. Also to be published in the *Communications of the Association for Computing Machinery* (Operating Systems Dept.)

For other references by D. L. Parnas, see P. J. Courtois *et al.*

Reddy, D. R., "Speech Input Terminals for Computers: Problems and Prospects," *Proc. 1970 IEEE International Computer Group Conference*, 282-289.

Reddy, D. R., L. Erman, and R. Neely, "The CMU Speech Recognition Project," *Proc. 1970 IEEE System Sciences and Cybernetics Conference.*

For other references by D. R. Reddy, see L. Erman *et al.*, U. Montanari *et al.*, and R. Neely *et al.*

Shaw, M., "360/Curse: Hymn of Hate," *Datamation*, April 1, 1971, 31.

Simon, H. A., "Gendai Soshiki no Kohon Dookoo," ("Basic Trends in Modern Organization"), *Soshiki Kagaku (Organizational Science)*, Summer, 1970, 44-52.

Simon, H. A., "Designing Organizations for an Information-Rich World," in Martin Greenberger (ed.), *Computers, Communications, and the Public Interest*, John Hopkins Press, Baltimore, 1971, 37-72.

Simon, H. A., "Information Storage as a Problem in Organization Design," *Ekomomiskt Firum*, Argang 33, 1970, 46-59. Goterborg: Handelshogskolans i Goterborg Studentkar.

Simon, H. A., "Information Storage as a Problem in Organi. ation Design," in Walter Goldberg (ed.), *Behavioral Approaches to Modern Management*, Vol. 1, Gothenburg Studies in Business Administration, Goteborg, 1970, 141-160.

Simon, H. A., *Ningen Kodo no Moderu*, Japanese translation of *Models of Man*, Dobunken Publishing Co., Tokyo, 1970.

Simon, H. A., and Y. Ijiri, "Effects of Mergers and Acquisitions on Business Firm Concentration," *Journal of Political Economy*, March/April 1971, 314-322.

Simon, H. A., and A. Newell, "Human Problem Solving: The State of the Theory in 1970," *American Psychologist*, Feb., 1971, 145-159.

Wulf, W. A., "BLISS: A Systems Programming Language," University of Pittsburgh Press, Sept.. 1970.

Zalcstein, Y., "On Star-Free Events," *Conference Record of the 11th Annual IEEE Symposium on Switching and Automata Theory*, Oct., 1970, 76-80.

Zalcstein, Y., and Y. Give'on, "Algebraic Structures in Linear Systems Theory," *Journal of Computer and System Sciences, 4*, 539-556.

Research Reports

Support for the following research work came largely from the Advanced Research Projects Agency (F 44620-70-C0107) and in part from the National Institutes of Health (MH-077722-02) and the National Science Foundation (GP-7064).

These reports are registered with the Defense Documentation Center. Accession numbers assigned as of July, 1971, are listed after the report titles.

Barbacci, M., H. Goldberg, and M. Knudsen, "C.ai (P.LISP)—A LISP Processor for C.ai," Computer Science Dept., CMU, June, 1971.

Krutar, R. A., "Conversational Systems Programming (or Program Plagiarism Made Easy)," Nov., 1970, unpublished.

Krutar, R. A., "Virtuality Is a Virtue," Nov., 1970, unpublished.

Lindstrom, G. E., ',Variability in Language Processors," July, 1970. (AD 714695)

Loveland, D., "Some Linear Herbrand Proof Procedures: An Analysis," Dec., 1970. (AD 717753)

McCracken, D., and G. Robertson, "C.ai—An L* Processor for C.ai," Computer Science Dept., CMU, April, 1971.

Montanari, U., "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," Jan., 1971.

Newell, A., J. Barnett, J. Forgie, C. Green, D. Klatt, J. C. R. Licklider, J. Munson, R. Reddy, and W. Woods, *Speech Understanding Systems: Final Report of a Study Group,* Computer Science Dept., CMU, June, 1971.

Newell, A., P. Freeman, D. McCracken, and G. Robertson, *The Kernel Approach to Building Software Systems.* Computer Science Dept., CMU, March, 1971.

Newell, A., D. McCracken, G. Robertson, and L. DeBenedetti, *L*(F),* Computer Science Dept., CMU, Jan., 1971.

Parnas, D. L., "Information Distribution Aspects of Design Methodology," Feb., 1971. (AD 719863)

Parnas, D. L., "A Paradigm for Software Module Specification with Examples," March, 1971.

Perlis, A. J., R. D. Fennell, F. J. Pollack, W. R. Price, and M. F. Rizzo, "Conversational Programming—APL: An Implementation in BLISS," Computer Science Dept., CMU, June, 1971.

Zalcstein, Y., "Locally Testable Events and Semigroups," Computer Science Dept., CMU, March, 1971.

Zalcstein, Y., "A Note on Fast Cyclic Convolution," Dec., 1970. (AD 717209)

RESEARCH REPORTS TO BE PRESENTED AT THE IFIP CONGRESS, LJUBLJANA, YUGOSLAVIA, AUGUST, 1971

Hansen, P.B., "An Analysis of Response Ratio Scheduling."

Parnas, D. L., "Information Distribution Aspects of Design Methodology."

Reddy, D. R., "Speech Recognition: Prospects for the Seventies."

Simon, H. A., "The Theory of Problem Solving."

Wulf, W. A., "Programming without the GOTO."

Colloquia

September 1970

"Storage Hierarchies—Heart of the Information Processing System"
Dr. E. W. Pugh, IBM

"Activities of PSAC—The President's Scientific Advisory Council"
Dr. H. Simon, Carnegie-Mellon University

October 1970

"Speech Synthesis"
Messrs. Coker and Umeda, Bell Laboratories

"Images from Computers"
Professor M. Schroeder, University of Goetingen

"Roles of Professional Society and Computer Science and the United Nations"
Professor C. C. Gotlieb, University of Toronto

"The Design of Operating Systems"
Professor B. Lampson, Berkeley Computer Corp.

"Operating Systems"
Professor B. Lampson, Berkeley Computer Corp.

"Graphics"
Dr. Ugo Montanari, Carnegie-Mellon University

"TSS"
Professor A. Kamerman, TSS Design Manager SDD, IBM and Adjunct Association

"TSS"
Mr. Nick King, TSS Productivity Manager, TSS Project, IBM

"Systems-Basic"
Professor T. Kurtz, Dartmouth University

"The Great Paper"
Professor P. Calingaert, University of North Carolina

"Great Machine"
Mr. R. Barton, University of Utah

November 1970

"Computer Science in the Soviet Union"
Dr. A. A. Ershov, Academy of Sciences, USSR, Novosibirsk

"Parallel Programming"
Dr. A. A. Ershov, Academy of Sciences, USSR, Novosibirsk

"Mathematical Semantics for Programming Languages"
Professor Dana Scott, Princeton University

"Operation of the Multi-Computer Lab at Livermore"
Dr. Sidney Fernbeck, Director of Computing of the Livermore National Laboratory of the University of California

December 1970

"Feature Extraction Techniques in Speech"
L. Rabiner, Bell Laboratories

"On Using Functional Analysis for System Design"
Dr. P. Freeman, Carnegie-Mellon University

"Complexity of Linear Inequalities with Application to Sorting"
Professor P. M. Spira, University of California at Berkeley

"OSL-TOOTH, An Operating System Language"
Mr. Peter Alsberg, University of Illinois

January 1971

"The Structures of a List Processing Computer"
Dr. Alan Kay, Stanford University

"Future Uses of Minicomputers"
Dr. Alan Kay, Stanford University

"Implementation of 'PMS' "
Michael Knudsen, Carnegie-Mellon University

February 1971

"PPL—An Extensible Language"
Dr. T. Standish, Harvard University

"A Natural Language Understanding System"
Dr. T. Winograd, Project MAC, MIT

"Survey of Graph Representations of Factual Information
Stu Card, Carnegie-Mellon University

"Graph Representation of Floor Plan Layouts"
Dr. J. Grason, Carnegie-Mellon University

Software implementation and hardware areas
Professor Dr. ir. Van Der Poel, Delft University

"Directed Graph Representation of Concepts and Experience"
Bill Mann, Carnegie-Mellon University

"Application to Electrical Networks"
Professor R. Duffin, Carnegie-Mellon University

"A Model for Functional Reasoning in Design"
Dr. P. Freeman, Carnegie-Mellon University

March 1971

"Graph Formulation of the Transportation Problem"
Professor G. Thompson, Carnegie-Mellon University

"DISCRETE-TIME Machines in Closed Categories"
Professor J. Goguen, University of Chicago

April 1971

"Toward a More General Theory of Data Structures"
Dr. D. Rine, West Virginia University

"The role of analytic models and simulation in the study of the feasibility of a circumferential communications network"
Peter Cook, IBM Watson Research Center

"The Irrelevance of Resolution"
Professor Seymour Papert, MIT

"A Set Theoretic Language for the Description of Algorithms"
Professor J. Schwartz, New York University

"Computers for Individualized Instruction"
Dr. R. Ferguson and Dr. K. Block, Learning Research and Development Center

"Flowchartable Recursive Specifications"
H. R. Strong, IBM

"Program Control Flow and Data Flow Analysis"
Fran Allen, IBM

"Planner"
Dr. Carl Hewitt, MIT

"Information Processing in Visual Perception"
H. A. Simon, Carnegie-Mellon University

"Rational Reconditioning of Polynomials"
Dr. M. Rabin, Hebrew University and IBM Research

"The Technology Chess Program"
J. Gillogly, Carnegie-Mellon University

Ph.D. Dissertations

The following persons have been awarded Ph.D.'s in Computer Science since the establishment of the Computer Science Department in 1965. immediately following each recipient's name is his position as of the 1970-71 school year.

Support for this work came largely from the Advanced Research Projects Agency under contract F-44620-70-C-0107. The accession numbers follow in parentheses after those dissertations which are registered as reports with the Defense Documentation Center.

Balzer, Robert M. (Systems and Communication Sciences), Computer Sciences Department, The RAND Corp., "Studies Concerning Minimal Time Solutions to the Firing Squad Synchronization Problem," 1966, Professor A. Newell. (AD 635056).

Berglass, Gilbert R. (Systems and Communication Sciences), Assistant Professor of Computer Science, state University of New York at Buffalo, "A Generalization of Macro Processing," 1970, Professor A. J. Perlis.

Caviness, B. F. (Mathematics), Associate Professor of Computer Science, University of Wisconsin, "On Canonical Forms and Simplification," 1967, Professor A. J. Perlis. (AD 671938).

Coles, L. Stephen. (Systems and Communication Sciences), Research Mathematician, Stanford Research Institute, "Syntax Directed Interpretation of Natural Language," 1967, Professor H. A. Simon. (AD 655923).

Darringer, John A. (Systems and Communication Sciences), Consultant, Logic Systems Design Department, N. V. Philips-Electrologica. Apeldoorn, The Netherlands, "The Description, Simulation and Implementation of Digital Computer Processes," 1969, Professor D. L. Parnas and Professor C. G. Bell. (AD 700144).

Earley, Jay. (Computer Science), Acting Assistant Professor, Department of Computer Science, University of California, Berkeley, "An efficient Context-Free Parsing Algorithm," 1968, Professor R. W. Floyd.

Ernst, George. (Systems and Communication Sciences), Associate Professor, Department of Computer Science, Computer Engineering Division, Case Western Reserve University, "Generality and GPS," 1966, Professor A. Newell. (AD 809354).

Evans, Arthur. (Mathematics), Lincoln Laboratories, Lexington, Mass., "Syntax Analysis by a Production Language," 1965, Professor A. J. Perlis. (AD 625465).

Feldman, Jerome A. (Mathematics), Associate Professor of Computer Science, Department of Computer Science, Stanford University, "A Formal Semantics for Computer Oriented Languages," 1964, Professor A. J. Perlis. (AD 462935).

Fikes, Richard E. (Computer Science), Research Mathematician, Stanford Research Institute, "A Heuristic Program for Solving Problems Stated as Nondeterministic Procedures," 1969, Professor A. Newell. (AD 688604).

Fisher, David. (Computer Science), Burroughs Corp., Paoli (Philadelphia), "Control Structures for Programming Languages," 1970, Professor A. J. Perlis. (AD 708511).

Freeman, Peter A. (Computer Science), Assistant Professor of Computer Science, Department of Information and Computer Science, University of California at Irvine, "Sourcebook for OSD—An Operating System Designer," 1970, Professor A. Newell.

Grason, John. (Systems and Communication Sciences), Assistant Professor of Electrical Engineering, Department of Electrical Engineering, Carnegie-Mellon University, "Methods for the Computer-Implemented Solution of a Class of 'Floor Plan' Design Problems," 1970, Professor H. A. Simon. (AD 717756).

Haney, Frederick M. (Computer Science), Manager of Design, Scientific Data Systems, El Cegundo, Cal., "Using a Computer to Design Computer Instruction Sets," 1968, Professor C. G. Bell. (AD 671939).

Iturriaga, Renato. (Computer Science), Director of the Computation Center, University of Mexico, Mexico City, "Contributions to Mechanical Mathematics," 1967, Professor A. J. Perlis. (AD 660127).

King, James C. (Computer Science), Research Staff, T. J. Watson Research Center, IBM Corp., "A Program Verifier," 1970, Professor R. W. Floyd. (AD 699248).

Lindstrom, Gary. (Computer Science), Assistant Professor of Computer Science, University of Pittsburgh, "Variability in Language Processors," 1970, Professor A. J. Perlis. (AD 714695).

London, Ralph L. (Mathematics), Associate Professor of Computer Science, University of Wisconsin, "A Computer Program for Discovering and Proving Sequencial Recognition Rules for Well-Formed Formulas Defined by a Backus Normal Form Grammar," 1964, Professor A. Newell. (AD 804036).

Manna, Zohar. (Computer Science), Visiting Researcher, Computer Science Department, Stanford University, "Termination of Algorithms," 1968, Professor R. W. Floyd. (AD 670558).

McCreight, Edward M. (Computer Science), Xerox Corp., "Classes of Computable Functions Defined by Bounds on Computation," 1970, Professor A. R. Meyer. (AD 693327).

Michell, James. (Computer Science), Xerox Corp., "The Design and Construction of Flexible and Efficient Interactive Programming Systems," 1970, Professor A. J. Perlis. (AD 712721).

Moore, James. (Systems and Communication Sciences), Research Associate, Carnegie-Mellon University, "The Design and Evaluation of a Knowledge Net for MERLIN," 1971, Professor A. Newell.

Mullin, James K. (Systems and Communication Sciences), Scientist, Division of Research In Epidemology and Communication Sciences, World Health Organization, Geneva, Switzerland, "A Computer Optimized Question Asker for Aiding Bacteriological Species Identification COQAB," 1967, Professor B. Green.

Parnas, David L. (Systems and Communication Sciences), Associate Professor of Computer Science, Computer Science Department, Carnegie-Mellon University, "System Function Description ALGOL—A Language for the Description of the Functions of Finite State Systems, the Simulation of Finite Systems, and the Automatic Production of the State Tables of Such Systems," 1965. (AD 467633).

Pfefferkorn, Charles. (Computer Science), Assistant Professor of Computer Science, Purdue University, "Computer Design of Equipment Layouts Using the Design Problem Solver (DPS)," 1971, Professor H. A. Simon.

Quatse, Jesse T. (Electrical Engineering and Systems and Communication Sciences), Vice President, Berkeley Computer Corp., "A Highly-Modular Organization of General Purpose Computers," 1969, Professor A. Newell and Professor C. G. Bell.

Quillian, M. Ross. (Psychology), Bolt, Beranek and Newman, Inc., "Semantic Memory," 1967, Professor H. A. Simon.

Shoup, Richard. (Computer Science), Xerox Research Center, Palo Alto, Cal., "Programmable Cellular Logic Arrays," 1970, Professor C. G. Bell. (AD706891).

Siklossy, Laurent. (Computer Science), Computer Science Department, University of Texas at Austin, "Natural Language Learning by Computer," 1968, Professor H. A. Simon. (AD 671937).

Standish, Thomas A. (Computer Science), Assistant Professor of Computer Science, Harvard University, "A Data Definition Facility for Programming Languages," 1967, Professor A. J. Perlis. (AD 658042).

Strauss, Jon C. (Systems and Communication Sciences), Director of Computing, Washington University, St. Louis, "Identification of Continuous Dynamic Systems by Parameter Optimization," 1965, Professor A. Lavi. (AD 660887).

Strecker, William D. (Electrical Engineering), Scientific Staff Engineer, RCA, "An Analysis of the Instruction Execution Rate in Certain Computer Structures," 1970, Professor C. G. Bell. (AD 711408).

Wagner, Robert A. (Computer Science), Assistant Professor of Computer Science, Cornell University, "Some Techniques for Algorithm Optimization with Application to Matrix Arithmetic Expressions," 1969, Professor A. J. Perlis. (AD 678629).

Waldinger, Richard J. (Computer Science), Research Mathematician, Information Science Laboratory, Stanford Research Institute, "Constructing Programs Automatically Using Theorem Proving," 1969, Professor H. A. Simon. (AD 697041).

Williams, Donald S. (Systems and Communication Sciences), Principal Member, Technical Staff, RCA Corp., "Computer Program Organization Induced by Problem Example," 1969, Professor H. A. Simon. (AD 688242).

Winikoff, Arnold W. (Systems and Communication Sciences), Director of Programming, Biomed Computer Services, St. Paul, Minn., "Eye Movement as an Aid to Protocol Analysis of Problem Solving Behavior," 1967, Professor A. Newell.